Canada $4.25
USA $3.50

# Transactor FOR THE Amiga

DECEMBER 1988: VOLUME 1, ISSUE 3

THE MAGAZINE FOR AMIGA PROGRAMMERS

- BreakPoint - third in a series on debugging *by Victor A. Wagner*
- Ray Tracing theory *by Cathryn Graham*
- Internationalizing your programs *by John Toebes*
- The ANSI standard for C *by Eric Giguere*
- A 68020/6881 board for the Amiga 1000! *by Brad Fowles*
- Ray Tracing with a 68020 system *by Louis Markoya*
- AmigaBasic 3-D plotting routines *by Anthony Bryant*
- BridgeBoard notes *by Dan Schein*
- **Plus** Reviews; regular columns by Larry Phillips, Don Curtis and Steve Ahlstrom; programming tips in *bits*; new products and more.

*Speed Reader* by Jo-Anne Park

# EXPLORE THE NEW WORLD

## Artificial Intelligence
## Expert Systems

You've heard the words. You probably know a bit about what they mean, too. You might associate them with white coats, Russian accents and shiny silver robots. What you might not know is that you can explore AI on your Amiga.

**MAGELLAN**™ by **Emerald Intelligence** offers performance and interface features UNAVAILABLE on the IBM-PC*, PS/2* or Apple Macintosh*. It was designed, developed and tested on the AMIGA for the AMIGA.

**MAGELLAN** turns *your* AMIGA into a powerful knowledge engineering workstation. It simplifies some of the complexities of artificial intelligence by using the power of the AMIGA. It is advanced enough to be simple.

The AMIGA's unique capabilities to support inexpensive desktop video, desktop publishing, audio/voice synthesis, extended memory, networking and high-resolution graphics make it ideal for advanced artificial intelligence development. Advanced to the state of being useful, not a graduate-school curiosity. Advanced enough to interface with the real world, and get to work. Advanced enough to be useful.

Use **MAGELLAN** to capture the expertise of your best salesperson, to help junior salespeople "learn the ropes". Build a system to diagnose sick AMIGA's and prescribe repair procedures. Weigh purchasing options in your job or at home. Evaluate performance of the hockey team. Drive animations with rules of behavior of the "actors" or objects. Create original computer art and music. Add real artificial intelligence to games to take them to the edge.

Best of all, use **MAGELLAN** to write programs. No BASIC, no "C", no compiler, no linking, no waiting. No kidding. With programs written with rules, changes can be made interactively — like BASIC — but much, much more efficiently. People don't think in flow charts, or "C" language structures or little curly brackets or parentheses. People think in terms of "IF — THEN". As a child your mother would say "If you take that cookie, then I'll . . ." Rules like this are how people work, how people decide. This is how computers will be programmed by our children.

Welcome artificial intelligence to the real world. Explore **MAGELLAN.**

M A G E L L A N

**List Price**
**$195**

Emerald
Intelligence

334 South State Street, Ann Arbor, Michigan 48104. (313) 663-8757.

*IBM-PC, PS/2 are registered trademarks of International Business Machines, Inc. *Apple Macintosh is a registered trademark of Apple Computer Corp. *AMIGA is a registered trademark of Commodore Business Machines, Ltd.

# THE SIXTH ANNUAL THE WORLD OF COMMODORE

**December 1 – 4, 1988**
**International Centre**
**Toronto**

Incredible bargains.
Amazing new hardware and software.
The excitement.
It all returns to Toronto for the sixth great year: Canada's annual computer extravaganza sponsored by Commodore Business Machines.

The 1987 show set world records – 42,000 showgoers made it the largest, best-attended Commodore show ever.

Commodore will be at the 1988 show with a giant display area for its exciting computers, accessories and software.

Thousands of brands will be shown and sold by other manufacturers, distributors and retailers.

Top experts in home, business and educational computing will lead seminars and demonstrations which are free with admission for all show visitors.

The Amiga, the C-64, the C-128, the Commodore PC line, all the major peripherals, programs, accessories – many more to be announced – they're all at the Sixth World of Commodore.

Get in on the excitement.

**Exhibitors, contact:**
**The Hunter Group**
**(416) 595-5906**
**Fax (416) 595-5093**

Produced in association with Commodore Business Machines

**Volume 1, Issue 3**

**Publisher**
Antony Jacobson

**Editors**
Nick Sullivan
Chris Zamara

**Assistant Editor**
Malcolm O'Brien

**Editorial Assistant**
Moya Drummond

**Customer Service**
Renanne Turner

**Contributing Writers**
Steve Ahlstrom
Scott Ballantyne
Charles B. Blish
Anthony Bryant
Jim Butterfield
Betty Clay
Don Curtis
Matt Dillon
Andy Finkel
Brad Fowles
Chad Innes
Eric Giguère
Cathryn Graham
Chris Gray
Charlie Heath
Perry Kivolowitz
Rico Mariani
Louis Markoya
Bryce Nesbitt
Rob Peck
Larry Phillips
Bob Rakosky
Dan Schein
John Toebes
Victor A. Wagner
David Wood

**Cover Artist**
Jo-Anne Park

# for the AMIGA Transactor
The Magazine for Amiga Programmers

## BreakPoint, Part 3 18
by Victor A. Wagner
How the advent of the CRT and cheap disk storage brought the art of debugging forward... and backward.

## 3-D Plotting for AmigaBasic 22
by Anthony Bryant
Short AmigaBasic procedures for plotting in 3-D, with some impressive sample programs.

## Ray Tracing Fundamentals 26
by Cathryn Graham
The theory of ray tracing, right down to the nitty-gritty. Complete with physics, math and C cod fragments for the total intellectual work-out.

## Turbo Tracing 32
by Louis Markoya
The bottom line: how much faster do speed-up boards improve ray-tracing performance? Here are som real examples with comparative times.

## LUCAS - A 68020/6881 board for the A1000 36
by Brad Fowles
You've heard about the incredible speed of accelerator cards, and probably about their incredible prices a well. Brad Fowles uses his 68020 board to introduce Public Domain hardware - just pay for the parts!

## The Final ARP.Library Tour 44
by Scott Ballantyne and Charlie Heath
ARP is the Amigados Replacement Project. Its authors document and discuss the new features in the AR library's latest revision: Process management, resident programs, and the file requester.

## The ANSI Standard for C 58
by Eric Giguère
Many C programmers are talking about the proposed ANSI standard for the language. Eric Giguer presents the most important points of the standard and shows how they'll affect your code.

## Internationalizing Your Programs 64
by John Toebes
How to make your programs useful to all users of the Amiga, no matter where they are in the world. In th process, your programs will become more flexible and easier to modify.

## Survival Guide to the BridgeBoard 71
by Dan Schein
Most-needed information about the hybrid Amiga.

# Departments and Columns


## Letters 5

## Bits 8

How not to blow up your Amiga 2000
Temporary disk labelling
Custom RAM disk icons
Suppressing and redirecting DOS requesters
Checking for I/O redirection
Easy multiple-raster allocation

## Amiga Dispatches 11

by Don Curtis

The fate of the A1000, ARP and AmigaDOS, and what's in Kickstart/Workbench 1.3.

## ACCESS - The best of non-commercial software 14

by Steve Ahlstrom

ACCESS votes for Access (the terminal program). This excellent terminal is investigated, followed by a look at Co-Pilot, the background utility with macro storage and lots of general convenience features.

## The ViewPort 16

by Larry Phillips

Let the Amiga make it on its own: Larry blasts the notion of boring business software on the Amiga.


# Reviews


## Manx Aztec C 3.6 and *sdb* 76

Manx's latest release adds complete support for *sdb*, the source level debugger. The first program of its kind for the Amiga, *sdb* almost makes debugging fun.

## ARexx 78

by Larry Phillips

ARexx really shows the benefits of Amiga's multitasking operating system. As more applications support ARexx, Amiga's user environment has the potential of going way beyond anything yet seen on a micro.


**About the cover:** *Speed Reader* by Jo-Anne Park

Jo-Anne Park is a Toronto artist who moved to the Amiga after using the Commodore 64 for a number of years, and is well known for her many outstanding Amiga pictures in the public domain. Admirers of Jo-Anne's work will recognize the subject matter as one of her specialties; others are Star Trek scenes and rock stars.

The picture was done using Deluxe Paint II on a 2.5 megabyte Amiga 1000 using 320 x 400 mode (interlace) with 5 bitplanes. Jo-Anne worked from plastic models and photos in the creation of the picture. She used an Easyl drawing tablet to enter the layout, and the mouse for the detailed work. The background was dithered with DPaint's 'fill' tool.


TransAmi's
phone number is:
(416) 764-5273
Mondays, Wednesdays
and Fridays ONLY

FAX: (416) 764-9262

**TOLL-FREE ORDER LINE**
**1-800-248-2719 Extension 911**
(for orders only; have your VISA or Mastercard number ready; available in the U.S. only)

Transactor for the Amiga is published bimonthly by Croftward Publishing Inc., 10-85 West Wilmot Street, Richmond Hill, Ontario, L4B 1K7. ISSN# 0838-0163. Canadian Second Class Mail Registration No. 7690, Gateway-Mississauga, Ont. US Second Class mail permit pending at Buffalo, NY. USPS Postmasters: send address changes to: Transactor, PO Box 338, Station C, Buffalo, NY, 14209.



Transactor Publishing Inc. is in no way connected with Commodore Business Machines Ltd. or Commodore Incorporated. Commodore and Commodore product names are registered trademarks of Commodore Inc.



**Subscriptions:** see order card at centre of magazine for complete ordering information and prices.

**Send all subscriptions to:** Transactor Subscriptions Dept. 10-85 West Wilmot Street, Richmond Hill, Ontario, Canada, L4B 1K7, (416) 764-5273. Note: Subscriptions are handled at this address ONLY. Subscriptions sent to our Buffalo address (above) will be forwarded to our Richmond Hill HQ. For best results, use the postage paid card at the centre of the magazine.

**Quantity Orders:** In Canada: Ingram Software Ltd., 141 Adesso Drive, Concord, Ontario, L4K 2W7, (416) 738-1700. In the USA: IPD (International Periodical Distributors), 11760-B Sorrento Valley Road, San Diego, California, 92121, (619) 481-5928; ask for Dave Bruescher. Quantity orders/enquiries are also welcome from computer/software distributors in the UK, Europe and Scandinavia. Please contact: T.G. Hamilton (W/S) Ltd., Tel: 021-742-5359; Fax: 021-742-2190; or contact Transactor (UK) direct at Unit 2, Langdale Grove, Bingham, Notts. NG13 8SR, telephone 0949-39380.

Editorial contributions are always welcome. Only original, previously unpublished material will be considered. Program listings and articles, including BITS submissions, of more than a few lines, should be provided on disk. Preferred format is Amiga 3.5-inch diskettes with ASCII text files. Manuscripts should be typewritten, double-spaced, with special characters or formats clearly marked. Photos should be glossy black and white prints. Illustrations should be on white paper with black ink only. IFF picture files or Aegis DrawPlus structured graphics files are preferred to hardcopy illustrations when possible.



All material accepted becomes the property of Transactor Publishing Inc., except by special arrangement. All material is copyright by Transactor Publishing Inc. Reproduction in any form without permission is in violation of applicable laws. Write to the Richmond Hill address for a writer's guide.

The opinions expressed in contributed articles are not necessarily those of Transactor Publishing Inc. Although accuracy is a major objective, Transactor Publishing Inc. cannot assume liability for errors in articles or programs. Programs listed in Transactor for the Amiga, and/or appearing on Transactor for the Amiga disks, are copyright by Transactor Publishing Inc. and may not be duplicated or distributed without permission.



**Production**
In-house with Amiga 2000 and Professional Page

Final output by Vellum Print & Graphic Services, Inc., Toronto

**Printing**
Printed in Canada by Bowne of Canada Inc.

# A Letter to our Readers

Well, for a bi-monthly magazine, we're not doing too well here: three issues in ten months. Subscribers will, of course, get all six issues they paid for with a one-year subscription, but they've had to be patient with us as the months dragged by without a sign of their next TransAmi. For this, we owe all of you some explanation.

First of all, you should know that despite rumours to the contrary, we're still in business and will continue to put out magazines. There was a time not too long ago, however, when we couldn't have said that with as much certainty. While Transactor - the magazine - had the capability of putting an issue to press on time (more or less), Transactor Publishing Inc. - the company - had a few problems. Like no money. You probably don't need a lesson in publishing to see that that could create a problem for the magazine.

While we had the prospect of late magazines on our hands, we had the question of survival itself to worry about as well. And we decided that releasing a late magazine is better than never releasing another magazine again, so we worked on getting the company on its feet and hoped that we could come out late and not disappoint too many readers in the process.

Well, we succeeded. An arrangement with Croftward Publishing Inc., a UK-owned publisher, has assured a stable base from which to produce magazines, so that Transactor can continue to provide the technical content we are famous for, while a large publisher with the associated resources can make it possible for us to do so. Additional staff and a new editorial schedule practically guarantees that each issue comes out on time. And best of all, while there will be more resources available to increase circulation and do all of those other wonderful publishing things, the arrangement is such that the editorial focus and the production quality of the magazine will remain unchanged - done by the same people and the same company as before. We were never very good at the business and publishing side of things anyway, so the new deal suits us just fine.

Editorially, things are looking just as rosy, as we continue to secure great articles by the best authors in the field. With all the new ideas and developments swirling around now, we don't stand in great danger of falling into a rut as far as content is concerned. If anything, we usually have the happy problem of deciding what to leave out. (Even this problem may eventually be solved once we encounter the even happier problem of too many pages of ads, leading to more pages in the magazine.)

We hope you enjoy this issue, and just as importantly, the ones that will follow in relatively rapid succession.

The Editors

# ReplyMsgs

**No extra hazard with PAK:** I must strongly object to the latter part of Steve Ahlstrom's review of PAK in your June issue. While his initial comparison of PAK and ARC is factual and fair, his last warning about PAK's "...biggest 'gotcha' [which] is potentially very dangerous" leaves something to be desired - namely, a warning that ARC is just as potentially "dangerous", as is ZOO. Without a complete explanation, the implication to a user who is not familiar with both PAK and ARC is that PAK is "dangerous", ARC is not. PAK is not a bit more "dangerous" than ARC or ZOO.

First, you can look into a PAK'd file to see what it contains before unPAKing it exactly like you can list the contents of an ARC'd file before unARCing.

Secondly, and most important, what in the world difference does it make whether you list the contents of a PAK'd, ARC'd, or ZOO'd file first? That just tells you the name that someone chose to give the program (which can of course be changed by anyone); it won't tell you whether that file in there called "NewDirUtil" is really a killer virus or not. The *only* way to positively ensure that a program isn't a Trojan Horse that will attempt to erase your hard disk is to completely disassemble it before running - which I doubt any sysop anywhere does. Does Mr. Ahlstrom really think that someone with a killer virus program would name it "KillerVirus", ARC it up and upload it so the sysop could list the contents and see that there's a file called "KillerVirus" included? Of course not; they would name it "NewDirUtil", ARC it and upload it, perhaps even with a fake doc file and fake source code.

Third, the point of the above is that it is just as easy to use ARC or ZOO for devious purposes as it is to use PAK. It is certainly possible for someone to hunkpad a file called "Surprise.PAK" and upload it, so that when you download it and try to unPAK it you are actually running the program instead of unPAKing a compressed file. But all that does is add one step on their end (the hunkpadding) and eliminate a step on the user's end (unARCing). It's just as easy for them to ARC a file called "Surprise", thus eliminating a step on their end (hunkpadding) and adding a step on the user's end (unARCing). If it turns out that "Surprise" is a killer virus then the only difference is that in one case it gets you when you type SURPRISE.PAK, while in the other case it gets you when you type RUN SURPRISE. Six of one, half dozen of the other. The comment "...if you don't want to blindly run an unknown file... PAK is not for you" does not make any sense. Unless you disassemble everything you unARC or unZOO, every single time you run a program you've downloaded you are blindly running an unknown file. All PAK does is remove the step of unARCing.

Despite your fine print disclaimer on your contents page, I feel you have a responsibility to correct this misconception about PAK. Opinions of columnists are one thing; misleading statements are another.

Joel Salazar, Tampa, Florida

*Steve Ahlstrom replies:*

*Joel,*

*You are correct. I was over-zealous. Without the PAK program itself the user has no idea of what is contained in a .PAK file. With ARChive or ZOO you can see a listing of the files contained within. Whether you can see the listing of files or not, makes no real difference, other than maybe a false sense of security to the novice user; you can still get bitten badly by a Trojan Horse type program. I will attempt to avoid such generalizations in the future.*

*Steve Ahlstrom, Littleton, Colorado*

**ARP Query:** The first issue of *Transactor for the Amiga* was quite impressive, even for a person like me who hasn't grown much beyond AmigaBasic. I presume that it will not be possible to use ARP with presently available DOS.bmaps. Does the ARP group plan to prepare an ARP.bmap? To what extent are the contents of *Transactor for the Amiga* (magazine and disk) in the public domain, freely copyable not distributable?

I look forward to reading about those two questions in one of the future issues.

Benito Lombardi, Pittsburgh, PA

*ARP doesn't come with a .bmap file, but it does have a .fd file from which you can generate the .bmap file using the BASIC ConverFD program in the BasicDemos directory on the Extras disk. The arp_lib.fd file can be found in the ARP release on the* ***TransAmi*** *disk for Volume 1, Issue 2. It is in the file INCLUDE.ARC, which is part of a larger ARC file called PROG.ARC.*

*To answer your second question, articles in* ***Transactor for the Amiga*** *are not in the public domain, and neither freely copyable nor freely distributable. In general, the programs are tutorial PD code, and you can pass them around as you please, though you may not duplicate the disk as a whole, upload the whole disk to a BBS, or in general spread it around as a collection or subcollection. If you ever are in doubt about what you may or may not legitimately copy, give us a call... we'll try to be accommodating.*

**Draco Update:** Great magazine! Keep it up. I read more of the text of *Transactor for the Amiga* than of any other magazine I subscribe to. Your highly readable listings, with fully preserved indentation, are a nice change to the "formatted" listings often found in magazines. I would suggest, however, that instead of vertically spacing them out to fill up a column, just put in a small cartoon, ad, or a simple doodle.

I would also like to take this opportunity to provide an update on the status of the Amiga Draco compiler. I have been working on it for the past month or so, and have made several improvements. 'Register' variables are now supported as are operator types, a couple of alternatives to 'pretend', and a new 'arbptr' type, which acts like ANSI C's 'void *'. I have also started adding optimizations to the compiler. It has a peepholer, and remembers register values. An output file change now allows a simple "Hello there, world!" program to be a 5K file instead of the previous 15K. Code size and speed are now quite comparable to that produced by Lattice C V4.0 (standard sieve is 40% smaller, 6% slower; registerized is 40% smaller and 3% faster). I have more to do yet, but there probably won't be much difference in that particular program. I plan on adding floating point support as soon as I get the new V1.3 IEEE library.

Chris Gray, Edmonton, Alberta

**My Compliments:** Just wanted to drop you a line to say "super job" on the first two issues of *Transactor for the Amiga*. Having been a subscriber to *Transactor* for several years, I expected no less in your Amiga edition. When you have such an outstanding group of contributing writers the magazine can't miss. Your magazines are definitely in the "read cover to cover" category and I have kept all of my *Transactors* for their reference value.

Please pass a "most well done" and my sincere appreciation to all your staff. I have nothing to add in the form of things you could do better - just keep evolving the way you have. Thanks for a great product.

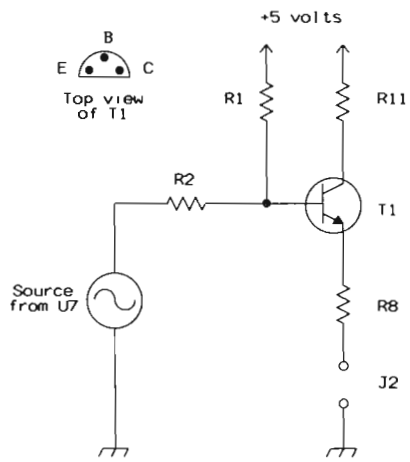
Gary L. Regan

**Bridgeboard Sound:** In Issue 2 of *Transactor for the Amiga*, Don Curtis wrote in his "Amiga Dispatches" column that the way to get sound from the Janus Bridgeboard was to connect a piezo buzzer to it. I wish to bring to his attention (and to the attention of *Transactor for the Amiga* readers) that, in accordance with the A2088 manual, one *can* attach a small 8-ohm speaker to the Bridgeboard. However, one must first repair what appears to be a design flaw in the Janus Bridgeboard. Fortunately, the repair is simple.

A couple of months ago, I became intrigued by the speaker problem. I took a close look at the Bridgeboard in the area around jumper J2 (where one connects the speaker), and it became apparent to me that something was wrong with the design. In the vicinity of J2, there are several discrete components that serve as an output buffer for the Bridgeboard's sound generator. I've enclosed a circuit diagram, Figure 1, of this output buffer. The components are arranged to form a textbook emitter-follower. Transistor T1 is fed the sound signal from a pin on chip U7 through R2, and it drives the speaker through R8. (The component labels in the diagram match those on the Bridgeboard.)

The culprit in the circuit (see diagram) is resistor R1, which is supposed to bias transistor T1. Before I repaired my Bridgeboard, R1 had a very low value -- it was either 10 or 100 ohms, I forget which. A low value for R1: a) improperly biases T1, and b) attenuates the signal from U7. Most people find that they cannot get an audible signal from an 8-ohm speaker attached to J2 because quite a bit of the signal is lost to AC ground through R1, and because transistor T1 isn't functioning properly.

The best way to repair the problem is to take advantage of your warranty and have a Commodore repair centre fix it. If your warranty has already expired or if you feel that you can't wait, simply replace R1 with a higher value resistor. Transistor T1 might also have to be replaced, but do this after you've replaced R1 and find that you still get no sound. I replaced R1 with a 3.3K quarter-watt resistor, a very common entity. The more sophisticated hacker might install a 1K resistor in series with a 10K vertical trim-pot (or some similar combination), the pot serving as a volume control. The transistor can be replaced with a general-purpose NPN BJT, but check the pinout first. Looking at transistor T1 from the top (i.e. with the leads facing away from the viewer), the order of the pins is emitter-base-collector (see diagram). I used a 2N2222-type transistor, which is packaged in a TO-1 case.




*Circuit diagram for the Janus bridgeboard speaker buffer*

Wayne Young, Toronto, Ontario

**Assemblers and Debuggers:** Okay, I know you're busy - so short and to the point:

Great first issue! Here is my subscription! Please review 68000 assemblers - the Metacomco product is slow enough for a leisurely coffee while it bumbles its way through 500 lines - and there is *no* debugger!

Any idea how to make the built-in ROM-Wack debugger work at something *other* than 9600 baud? I'd like to use my C128 as a debug terminal but 4800 baud is *really* pushing it - and 9600 is beyond the hardware!! Curious that the Preferences are not used by the debug, eh? Or is it only for *user* friendliness - not *programmer* friendliness?

Bryce Nesbitt's "Hexadozen Amiga Programming Tips" was super! Please print another $186A0!

Peter Wyspianski, Kingston, Ontario

*Peter, as far as we know, ROM-Wack works at 9600 baud, period. Any information to the contrary would of course be most welcome.*

*However, it is not true that 9600 bits per second is beyond the C128 hardware. We have heard lately that a Toronto-area developer has come up with a terminal program for the 128 that achieves 9600 bps transfer rates by resorting to wholesale replacement of the system serial routines. The program is a yet-to-be released commercial product, but our impression is that it is close to release.*

**A Modula-2 reading list:** As an Amiga owner, active AUG member and fan, I welcome your new magazine. It was like a breath of fresh air to read your first issue. Here was something that counteracted the saccharin, cutesy and klutzy, know-nothing commercialism of *AmigaWorld*! The excellent *Ami Project* magazine that was published here in Ohio lasted only for 6 or 7 issues (I suspect that the publisher got tired, rather than broke). I certainly hope that you fare better. I mention this because I really was about to sign for a three-year subscription with disks when I read Kivolowitz's article and the previous adjectives came to mind... plus a choking dose of ego... it would be nice to make out a bit of content behind his image. I actually turned pages to see if you had appended his ads to the article. In general, the articles were very well done, but do you realize that (except for mentions by Nesbitt and Toebes) there was nothing on graphics or music? To paraphrase Tigger & Pooh... those are what Amigas do best!

It was encouraging that Phillips mentioned Modula-2 and Icon. We now have three commercial versions and one PD version of Modula-2 on the Amiga. I use it in my programming courses and, when I mentioned that to publishers, I received about two dozen books on Modula-2 (a book list is enclosed).

Has anyone considered combining the Amiga's multi-tasking and Modula-2's concurrency monitoring abilities?

Please, let's have more space for and by Amiga amateurs (lovers!) and a bit less by self-promoting commercial C-hackers. Again, I wish you well, but I'll opt for a one-year trial to help prevent complacency and encourage responsiveness to subscribers. (You know... "keep 'em lean'n'mean.")

Here are some text/reference books (26) which indicate the great interest in Modula-2 just within the last three years:


Biedler, J. & Jackowitz, P., *Modula-2*, Prindle, Weber, Schmidt 1986, 347pp. (paper).

Christian, K., *A Guide to Modula-2*, Springer-Verlag 1986, 436pp.

Cleaves, R., *Modula-2 for Pascal Programmers*, Springer-Verlag 1984, 145pp. (paper).

Feldman, M.B., *Data Structures with Modula-2*, Prentice Hall 1988, 320pp.

Ford, G.A. & Wiener, R.S., *Modula-2, a Software Development Approach*, J. Wiley & Sons 1985, 400pp. (paper).

Greenfield, S.B., *Invitation to Modula-2*, Petrocelli Books Inc. 1985, 245pp.

Joyce, E.J., *Modula-2: A Seafarer's Guide and Shipyard Manual*, Addison-Wesley 1985, 288pp. (paper).

Kaplan, I. & Miller, M., *Modula-2 Programming*, Hayden Book Co. 1986, 240pp. (paper).

Koffman, E.B., *Problem Solving and Structured Programming in Modula-2*, Addison-Wesley 1988, 750pp. (paper).

McCracken, D.D. & Salmon, W.G., *A Second Course in Computer Science with Modula-2*, J. Wiley & Sons 1987, 450pp.

Messer, P. & Marshall, I., *Modula-2: Constructive Program Development*, Blackwell Scientific Publications 1986, 544pp. (paper).

Moore, J.B. & McKay, K.N., *Modula-2 Text and Reference*, Prentice Hall (Reston) 1987, 419pp. (paper).

Pinson, L., Sincovec, R., Wiener, R., *A First Course in Computer Science with Modula-2*, J. Wiley & Sons 1987, 491pp.

Sale, A., *Modula-2, Discipline and Design*, Addison-Wesley 1986, 452pp. (paper).

Sawyer, B. & Foster, D., *Programming Expert Systems in Modula-2*, J. Wiley & Sons 1986, 201pp. (paper).

Schildt, H., *Modula-2 Made Easy*, McGraw-Hill (Osborne) 1986, 377pp. (paper).

Schildt, H., *Advanced Modula-2*, McGraw-Hill (Osborne) 1987, 379pp.

Sincovec, R.F. and Wiener, R.S., *Data Structures using Modula-2*, J. Wiley & Sons 1986, 500pp.

Stubbs, D.F. & Webre N.W., *Data Structures with Abstract Data Types and Modula-2*, Brooks/Cole 1987, 556pp.

Terry, P.D., *A First Course in Programming with Modula-2*, Addison-Wesley 1987, 384pp. (paper).

Thalmann, D., *Modula-2: An Introduction*, Springer-Verlag 1985, 292pp. (paper).

Tucker, A.B., *Computer Science: A Second Course using Modula-2*, McGraw-Hill 1988, 405pp.

Walker, B.K., *Modula-2, Programming with Data Structures*, Wadsworth, 434pp.

Welsh, J. & Elder, J., *Introduction to Modula-2*, Prentice Hall (to appear) 360pp. (paper).

Wirth, N., *Algorithms and Data Structures*, Prentice Hall 1986, 288pp.

Wirth, N., *Programming in Modula-2*, Springer-Verlag 1985, (3rd Ed.) 176pp.


Roger Messick, Buffalo, New York

*Thanks for passing along the book list, Roger - we're sure many readers will find it useful. Thanks also for your charitable interest in keeping us lean and mean. We liked Perry's article, though.*


**68000 DISASSEMBLY**

OTG Software

**ANNOUNCING . . . DSM VERSION 1.0d**

**DSM is a full-featured disassembler for the Amiga. Check out these features and you'll see why programmers agree, "DSM is the best disassembler currently available for the Amiga, bar none."**

- **DSM disassembles any Amiga program that does not use overlays.**
- **DSM produces assembler-ready output that is 100% compatible with the Amiga assembler, assem. Excellent results can also be obtained using the C.A.P.E. assembler from Inovatronics.**
- **DSM features true text detection.**
- **Improved disassembly is supported using the built-in expert system mode.**

**If your local Amiga dealer doesn't carry DSM, send check or money order to:**

**OTG Software**
**200 West 7th Street Suite 618**
**Fort Worth, TX 76102**

**Price $67.50 US**

**TX residents add 7.25% sales tax.**

# b i t s

*Got an interesting programming tip, a short routine, or an unknown bit of Amiga trivia? Send it in - if we use it in the bits column, we'll credit you in the column and send you a free one-year subscription to Transactor for the Amiga.*

**How *not* to blow up your new Amiga 2000**
**Jeff Tyler**

In the *Byte* product preview and several other sources it was reported that the wizards of West Chester had cleaned up the Amiga 2000's serial and parallel ports and removed dangerous voltages, etc. Well... not really. The parallel port is now a DB25 female and the pinout is much more standard *à la* Big Blue but some weird things remain. The infamous Amiga +5 VDC is still there (found on pin 14) and reset has been moved to pin 16 which is also standard but will not work with Amiga 1000 custom printer cables (to say nothing of the sex confusion). If you want to use an existing Amiga 1000 printer cable, move pin 19 to pin 16 and plug in a male/male gender bender and you're on your way.

The serial port is another matter, a much more dangerous one. The serial port is a DB25 male with all the usual RS-232 leads in the normal place but with +/-12 VDC on pins 9/10 as well (similar to the Amiga 1000). If you short pins 9/10 I guarantee that you will ZAP your power supply!! Not just a section but the entire thing. Some null modem cables and adapters do just that. Ask the guy that did it. So... cut pins 9/10 dead at your female cable, the end that plugs into the 2000. Shorting the +/-12 is much more dangerous than shorting the +5, especially at power up, as it draws a dead shot surge through the secondary of the transformer with fatal results.

Shorting the +5 on the other hand normally only results in a blown fuse - if you're lucky. So much for standard connectors with all dangerous voltages removed. One expects *Byte* to be more accurate but it was just a product review. In fairness to CBM, the Amiga 2000 book does indeed show all of this, but doesn't warn of the possibility of a fatal short.

**Temporary Disk Labelling**
**Neal Bridges**

The 3.5" disks are terrific; but for quick, temporary labelling of a disk, the sticky labels can be a pain. Fortunately, I have discovered an easy method of temporarily labelling disks. It's a product called "Taperaser" from the Dixon Pencil Company, and was originally designed to correct typewriter mistakes. It comes in the form of a yellow paper tape with white powder on one side. To write on a plastic disk shell with it, place a strip of it (white side down) on the disk and write the desired information on the yellow side with a ballpoint pen. The resultant white lettering is highly visible, and can be removed cleanly with a pencil eraser.

**(Not) another Startup-Sequence batch file!**
**Anthony Bryant**

This one displays a custom command batch file, to either add to the existing sequence or Execute on its own.

Using IconEd (v1.2), create your own RAMDisk icon. The easiest way is to load SYS:Disk (.info will be added), change it, then "Frame and Save" it as SYS:RAMDisk (.info will be added).

Now Execute this RAMdisk-Sequence batch file:

```
Copy RAMDisk.info to RAM:Disk.info
MakeDir RAM:C
Copy :C/CD   to RAM:C quiet
Copy :C/Dir  to RAM:C quiet
Copy :C/List to RAM:C quiet
Copy :C/Type to RAM:C quiet
Copy :C/Info to RAM:C quiet
etc.
Path Add RAM:C
```

The first Copy command creates the RAM Disk and displays your custom icon. Obviously, you can choose whatever AmigaDOS 1.2 commands you like within the batch file. The Path command avoids the use of lots of Assigns. DOS now searches RAM:C *before* it searches SYS:C. Note that v1.2 Startup-Sequence already has two Path Add commands which cause it to look for commands in SYS:System and SYS:Utilities if not found in the 'normal' SYS:C directory.

### Checking for I/O Re-direction

It's often necessary for a program to know whether it's being run with its input (stdin) redirected. For example, you might want to suppress prompts if the input data is coming from a file. The DOS function IsInteractive() tells you this for programs executed directly, but always returns FALSE for programs that are RUN. If a program reading from stdin is RUN without input redirection, you need to know that there's no input channel, otherwise the program will hang when it attempts to read. Here's how to check for all cases:

```
#include <libraries/dosextens.h>

struct Process *proc, *FindTask();
long IsInteractive();
BPTR DOSstdin, terminal, Input();
struct CommandLineInterface *cli;

proc     = FindTask(0L);
terminal = Input();
cli      = (struct CommandLineInterface *)BADDR(proc->pr_CLI);
DOSstdin = cli->cli_StandardInput;

if (IsInteractive(terminal))      puts("Input is interactive");
else if (DOSstdin != terminal)    puts("Input is coming from a file");
else                              puts("No input channel");
```

### Easy multiple raster allocation

To create a BitMap, you have to fill in a 'BitMap' structure appropriately, including pointers to the bitplanes themselves, which you usually allocate with AllocRaster(). After calling InitBitMap() to initialize the width, height and depth information in the structure, you can use the handy AllocRasters() routine below to allocate all bitplanes and put pointers to them in the BitMap structure. Just pass AllocRasters() a pointer to your BitMap structure and it does the rest.

If any allocation fails, any previously allocated planes are freed, and the function returns FALSE (0). Make sure you check the return code from AllocRasters() before continuing execution, because if your program tries to use the BitMap and the bitplanes haven't been allocated, look out! The companion routine FreeRasters() is used to later free all allocated bitplanes. These are useful routines to have around for any graphics programming you might be doing where you need your own BitMaps.

```
/* AllocRasters() attempts to allocate ALL rasters for a given BitMap.
   If any allocation fails, it returns a zero, else it returns 1.
*/
AllocRasters (bitmap)
register struct BitMap *bitmap;
{
register int i;

   for (i = 0; i < bitmap->Depth; ++i)
      if ( !(bitmap->Planes[i] =
            AllocMem((long)(bitmap->Rows) * bitmap->BytesPerRow, MEMF_CHIP)) )
         return FreeRasters(bitmap);

   return 1;
}

/* FreeRasters() frees bitplanes allocated by AllocRasters().
   It always returns a zero.
*/
FreeRasters (bitmap)
register struct BitMap *bitmap;
{
register int i;
long size;

   size = (long)bitmap->Rows * bitmap->BytesPerRow;
   for (i = 0; i < bitmap->Depth && bitmap->Planes[i]; ++i)
      FreeMem(bitmap->Planes[i], size);

   return 0;
}
```

### Suppressing and Redirecting DOS Requesters

The DOS requesters that come up asking the user to insert a certain volume are usually useful for an application, since it just has to ask for a volume without worrying about whether it's mounted or not. However, an application on a custom screen might want the requesters to come up on that screen, so that the user isn't faced with the Workbench screen unexpectedly when a requester pops up. There are other situations where you don't want the requesters to come up at all, such as when using a CLI from a remote terminal: if a requester pops up, you have no way of making it go away, and you're stuck. Another reason for not wanting them is that some programmers would rather just get an error message in their CLI window than have to grab for the mouse every time they type a bad volume name.

You can suppress or redirect DOS requesters for any CLI by changing the WindowPtr field in the CLI's Process structure, as follows:

```
struct Process *proc;
struct Task    *FindTask();
struct Window  *window;

  proc = (struct Process *)FindTask(0L);

  proc->pr_WindowPtr = (APTR)window;   /* DOS requesters go to window */
  proc->pr_WindowPtr = (APTR)-1L;      /* requesters are suppressed   */
  proc->pr_WindowPtr = NULL;           /* requesters to CLI window    */
```

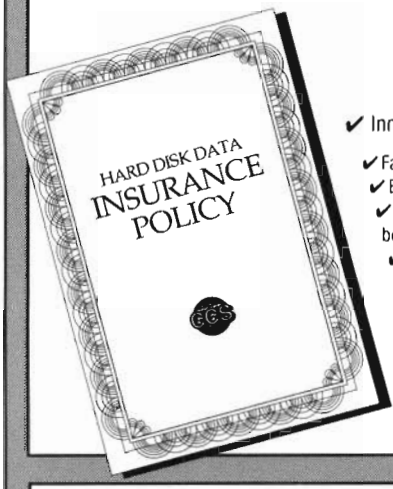
Note: If you change the status of the WindowPtr field in this manner, you should save its original value so that you can restore it after your program exits. You don't want to change the user's CLI environment on him unexpectedly!

FOR THE AMIGA

***Protecting your valuable hard disk files is easier and faster than ever before!***

# QUARTERBACK™

*The **FASTEST** Hard Disk Backup Utility!*




***Backup to or restore from:***

✔ Floppy Disks ✔ Streaming tape (AmigaDOS-compatible)
✔ CLtd's Konica 10.7MB high-density floppy drive
✔ Inner-Connection's Bernoulli drive ✔ ANY AmigaDOS-compatible devise

✔ Fast backup - 20MB in 30 minutes or less ✔ Uses two floppy drives (if available) for backup/restore with automatic switching ✔ Builds, sorts and displays catalog of files and subdirectories ✔ Provides FULL/Subdirectory/Individual file backup/restore ✔ Includes or excludes files by name (with wild cards), file date, or archive bit ✔ Calculates the number of floppies you'll need before you start ✔ Handles files of unlimited length, unlimited subdirectories and unlimited files per subdirectory ✔ Automatically formats diskettes with no delay as it writes ✔ Sequentially numbers and date/time stamps backup diskettes ✔ Checks the sequence number and date/time stamp of each diskette before restoring files from it ✔ Restores original date/time stamp, file notes, and protection bits on both files and subdirectories ✔ Runs with Workbench or CLI ✔ Produces backup/restore report to disk or printer ✔ Beeps for floppy change ✔ Accepts CLI parameters and batch command files ✔ Detects bad disks during backup or restore ✔ Convenient/user friendly error recovery ✔ Multitasking ✔ Runs in 512K ✔ No copy protection ✔ Works with all AmigaDOS compatible hard disk drives.

**Only $69.95** Plus $3.00 shipping and handling, CA residents ass 6% sales tax.

## DISK 2 DISK™

### Convert C64/C128 Files to the Amiga!

DISK-2-DISK™ makes it easy and convenient to transfer C64/C128 files to and from the Amiga! DISK-2-DISK programs the Amiga model 1020 external 5.25" disk drive to read and write 1541/4040 and 1570/1571 disk formats including 1541 "flippies".

✔ Converts Commodore/PET ASCII to AmigaDOS standard ASCII and vice versa ✔ Transfers word processing text files (such as PaperClip, SpeedScript and Pocket Writer) to and from the Amiga for use with popular Amiga word processors ✔ Includes 3 public domain programs for converting C64 Koala, PrintShop and Doodle files to IFF format ✔ Finds and flags dialect differences between Commodore Basic and Amiga Basic files ✔ Provides VALIDATE BAM and CHECK DISK utilities (VALIDATE BAM verifies the directory structure of the 1541/1571 diskette; CHECK DISK reads every block of a 1541/1571 diskette to detect diskette errors).

*DISK-2-DISK requires the Amiga model 1020 5.25" disk drive.*

**Only $49.95**
Plus $3.00 shipping and handling
CA residents add 6% sales tax.

## DOS 2 DOS™

### DOS-2-DOS transfers MS-DOS and Atari ST files to and from AmigaDOS!

**DOS-2-DOS** version 3.0 permits access to any MS-DOS volume available via AmigaDOS, including MS-DOS partitions on hard disks and MS-DOS volumes on LANS or SCSI networks.

✔ Supports single and double sided 5.25-inch as well as 3.5-inch 720KB MS-DOS diskettes ✔ Reads/Writes 3.5-inch Atari ST diskettes (GEM format) ✔ Reads a variety of 5.25-inch MS-DOS floppy formats via the CLTD Konica high-density floppy drive ✔ Converts ASCII file line-ending characters and provides Wordstar compatibility ✔ Supports full directory path names, with wild cards in the file names ✔ Allows selection of MS-DOS and AmigaDOS subdirectory and displays sorted directory listing ✔ Formats 3.5-inch and 5.25-inch MS-DOS diskettes and Atari ST diskettes ✔ Provides duplicate file name detection with query/replace options ✔ Provides TYPE and DELETE commands ✔ Permits renaming of files where file name restrictions occur ✔ Remains resident to permit AmigaDOS disk swapping.

**Only $55.00**
Plus $3.00 shipping and handling
CA residents add 6% sales tax

CCS

# Central Coast Software™

268 Bowie Drive, Los Osos, CA 93402 • Telephone: 805 / 528-4906 • FAX: 805 / 528-3138

*DEALER INQUIRES WELCOME*

VISA

# Amiga Dispatches

## *Whither the A1000, and what's new in WB1.3?*

**by Don Curtis**

*Don Curtis is a police officer in Denver, Colorado, who has for the last two years been assigned to program design and development along with system design and maintenance of 10 AT&T Unix 3B2 computers. In his spare time, Don is an assistant sysop on CompuServe's Amiga forums.*

Before we get started, a bit of a correction from last time. I'm so used to specifying RAM usage in bytes that, without thinking, I said last issue that the rumoured resolution of 1024x920 possible for an A3000 would eat up about a Mega*byte* of RAM per bitplane. Well, obviously that's not true, it would eat up about a Mega*bit* of RAM per bitplane - one eighth the amount. I knew what I meant, I just didn't say it right!

Well, the long cold winter is now over, and by the time you read this, summer ought to be in full swing. Summer is usually a slow time for computer-related activities because there are so many other things to do. With that in mind, I thought I'd heat things up with a controversial subject. The fate of the A1000...

What is going to become of the A1000? Is it obsolete? Is it worth buying, keeping, having? This is a touchy subject but, the way I see it, the answers are: nothing, probably and yes. Okay, before you start turning up the flames, let me explain.

Nothing will become of the A1000 - it will remain essentially what it is today, with little or no change possible. The Amiga hardware is evolving in a manner that is incompatible with the A1000. The new Fat Agnus won't fit, and a hardware hack to adapt it to the A1000 is probably a waste of time.

Both the A500 and the A1000 share the same side expansion port, but with major physical differences. The A500's port is on the left side of the machine and reversed front-to-back from the one on the A1000. Not only that, but they are at different heights. Commodore has sold many more A500s than it has sold A1000s and, as a result, the 'standard' configuration for a side expansion add-on is now the A500. This means that side expansion boxes will have to be raised up to fit into the A1000. That's not a major problem in itself, but most expansion add-ons would be designed to extend towards the back of the A500 to maintain a neat, clean appearance. With the reversed configuration of the A1000, that means that the extension would be towards the front, and would probably cover up the mouse port. If that's not enough, add-on cables to another device, like a hard disk, would come out the front of an add-on mounted on the A1000. All in all, those differences mean that most add-ons designed for the A500 won't work on an A1000.

The supply of add-ons designed for the A1000 will soon start to dry up. The machine hasn't been in production for over a year now. Hardware manufacturers have been seeing sales of A1000 add-ons slow to a crawl. As the stocks of add-ons drop, manufacturers will have to consider whether it's worth it to make new production runs. Among the things to be considered are the potential sales. A manufacturer who has only sold 100 of a certain item in the last year would be hard pressed to come up with a good reason to make a new production run. To be economical, a production run should be at least 1,000 pieces, which in this example means a ten-year supply of parts for a machine no longer in production - not a very smart idea.

A few items *can* be made in small production runs. Michigan Software started out making the Insider by hand, and I presume they could continue to supply the Insider in very small lots as needed. But again, for the most part, very few items will be available once the present stocks dry up.

A few items will be designed so that they can be adapted to either the A500 or A1000. For a while, this will mean that there will be an ample supply of these items. But again, as sales of the A1000 configuration slow down due to lack of demand, less and less products will be designed to be 'multi-machine'. Remember, there aren't very many A1000s out in the world. Pass-through devices don't always work on the A1000 when you have more than one hooked up. Even if they did, stacking three or four add-ons can be very awkward. The upshot is that you have a limited number of machines, each able to accept a limited number of devices. As the owners of the A1000 *do* expand their machines, the number of potential buyers for add-ons drops. Fewer potential buyers translates into fewer sales, and less interest on the part of manufacturers.

There will be exceptions to all of the above. Some small company or individual will come up with some clever devices for the A1000, but you won't see large selections of add-ons for the A1000 in the future. When expansion capability stops, the computer will be at a dead end.

Is the A1000 worth buying? It probably is, because if it fits your needs, it can be had at a very reasonable price. No computer is ever truly dead as long as it still works. When the A1000 came out, it was new and exciting and did lots of things. It can still do all those things; if that's all you want from your computer, the A1000 will be perfect for you. Nothing says you have to have the latest and greatest. If your use of a computer is limited to things like telecommunications, word processing, record keeping and some games, then an older, less expensive model may very easily fill your needs. I know lots of folks who use eight and ten year old 8-bit computers with 48K of RAM and are perfectly happy. There will also be a large group of folks who will be perfectly happy with a three or four year old 16-bit computer with 512K of RAM and a couple of disk drives.

And yes, the A1000 is worth having. As stated before, it is still very much an Amiga. It can still be expanded to a 'full-blown' system that would have cost tens of thousands of dollars in the past. It still runs tons of good software. If I were given a choice between a brand new IBM PC and an Amiga, I'd take the Amiga any day. There is no question in my mind that for the types of work I do on a computer, the Amiga is the best machine for me and the A1000 is no exception. If you don't need all the newest and fanciest gadgets, you can be happy for years to come with the 'Original Model'. Anyway... it still has the checkmark!

**ARP and AmigaDOS...**

Another bit of controversy has flared over the networks recently. Charlie Heath (of Microsmiths) and others have recently put together the ARP replacement command set for the Amiga. The AmigaDOS Replacement Programs command set was written in assembly and has more features and flexibility than the original AmigaDOS command set. Charlie and his group have made ARP freely distributable, and posted it to most of the networks. On top of that, they have offered it to Commodore for their use to replace the original commands. And Commodore has rejected the offer... for now.

There were several reasons given for the rejection, among them the fact that ARP came late in the Workbench 1.3 testing cycle and thus would have required a complete new test period. Commodore people have also objected that ARP is less maintainable than Commodore's command set, which is written in C, and that assembler isn't necessarily appropriate for the command set.

The controversy arose over the last two objections. Since Commodore uses a team of programmers, and members of the team can change, they feel that a language like C is more appropriate from a maintenance viewpoint. Without getting into the 'religious' argument of High Level Language versus assembler, I have to say I agree with their viewpoint. Having worked on several team projects, it was very easy to interface my portion of the code with the rest using C. Using assembler would have been a much harder task.

Now, I freely admit that I'm not an expert or even proficient in assembler, but I have worked with it. One of the original projects I got involved in was written entirely in 6502 assembly. I was simply beta-testing this particular program, but every time I found a bug, the programmer had to pore over a program listing that looked like a Manhattan phone book. Even though the programmer was an expert in 6502 assembly and had well commented code, it would take him about half an hour going back and forth between code fragments to find the bug. Our next project was an equivalent program, but this time written in C. I was one of the programmers on this project and, when a bug was reported, it would typically take about ten minutes to find it and fix it. Any of the programmers on the project could pick up anyone else's module and quickly figure out what the other person was trying to do.

High Level Languages have another plus in the maintenance area - much more of the code will fit on a screen at once. It may take a full screen of assembly code just to set up to do an open call, whereas one screen of HLL can not only set up the call, but also open the file (with error handling), read, do data manipulation and then close the file. It's much easier to comprehend a program when you can see more of what's going on.

On the question of the appropriateness of assembly for the command set, I again tend to agree with the folks at Commodore. I think most programmers would agree with the proposition that it generally takes longer to write a program in assembler than it does in a HLL. I think most programmers would also agree that programs written in assembler run faster. The question reduces to this: Is it cost effective to spend from three to ten times the amount of time (which translates into money) to pick up a bit of speed on a command that is rarely used? Is it really that important for a JOIN of two files to run in two seconds rather than four seconds? If I do a LIST, and the screen scrolls by so fast that I have to pause it, is it worth the time and effort to speed it up even more?

Assembly is appropriate for things like the kernel (Exec) and the basic user interface (Intuition). On this level, every bit of speed is needed. The routines are called often, and slowdowns can mean significant delays in overall use of the machine. If it takes twice as long to switch from process A to process B, the overall effect is that both processes run at a slower speed. That can be significant. If it takes twice as long to do an ECHO command during your startup script, it really isn't a big deal.

The biggest slowdown in the command set is the time it takes to load the program from disk into memory and start it up, not the execution time. Rather than spend the time to start all over and recode the command set into assembler, I would much rather see Commodore take the time ensure that their current code is truly re-entrant, so it may be made resident.

**KS/WB v1.3**

Hopefully, v1.3 of Kickstart/Workbench will be officially out by the time you read this, but in case it isn't, and/or you

missed our last issue, here are some more of the goodies *expected* to be in the release version. Note: None of these items is guaranteed to be in the release version but most of them should be. Keeping that in mind, let's look at the major changes.

First and foremost (in my mind) is the Fast File System (FFS). Without going into all the details, what CBM has done is trade a bit of recoverability for significantly faster disk access. Under the old file system, a 512-byte data block contained only 488 bytes of actual data. The other 24 bytes were used by DOS to assist in recovery. This meant that DOS had to read each block separately into a private buffer, strip off the 24 bytes of file system overhead, and only then move the 488 data bytes to their real destination in memory. The FFS has none of this overhead - a 512-byte data block contains a full 512 bytes of data, and thus can be moved directly into memory. Speed-ups of six to ten times over the old file system have been reported. Other changes include the fact that a script file can now be placed in the logical S: directory and will be automatically executed when a CLI is opened. This script can be used to alias commands for the Shell.

The Shell? Oh yes, that's a replacement for the normal CLI that you can use if you wish. It's a handler (L:Shell-Seg) that can be made resident under the name CLI. The Amiga shell,in conjunction with an optional new console (CON:) handler (L:NewCon) now allows you to alias command names to pretty much whatever you'd like. Also included is an editable command line, command history, and support for memory resident commands if the command is 'Pure' (a Pure file is both re-entrant and re-executable). The Shell also allows you to put your current directory path name in your CLI prompt.

Even more important, the Shell supports the use of the 'script bit', a new addition to the set of flag bits recognized by the Protect command. You can set bits specifying that a file is hidden (from directory listings), that it is a script file, or that it is a pure (residentable) file. Having the script bit set tells the new console handler that the file is an executable script file, so you no longer need to type "Execute <filename>" - simply type the file name alone, just as you would with a program file, and the handler will invoke Execute for you automatically. All in all, the user interface to the CLI has been greatly improved.

You'll also find a new Recoverable Ram Disk (RAD:). This is a device file (DEVS:ramdrive.device) that must be mounted and formatted prior to use; however, once it's created, it will survive a warm start. Not only that, you can auto-boot from it.

Auto-boot is also available from a hard drive under certain conditions. Hard drives will require an auto-boot ROM. The current A2090 doesn't have this ROM, but its replacement, the A2090A, will. If you have an A500 or A2000, you'll also need new Kickstart ROMs to allow auto-boot. On the A1000, you still must use a v1.3 Kickstart disk on a cold-start, but from then on, you can warm-start via auto-boot from either the hard drive or Recoverable Ram Disk.

Pipes have been added as a handler (L:Pipe-Handler) and must be mounted prior to use. Using a pipe, you can take the output of one command (whatever the command writes to its standard output) and 'pipe' it to another command (which reads the piped data as its standard input).

The printer handlers have been vastly improved. Along with the appropriate changes in Preferences, the printer drivers are much faster, handle colour output much better, and support more features. You will also be able to use the new CMD command to re-direct the output of the printer devices to a file, rather than the I/O ports themselves.

On top of these major changes, there are other enhancements:

An Aux-Handler has been added, similar to the SER: handler except that it must be mounted and uses unbuffered I/O. It allows the use of the serial port for a remote terminal.

The Clock now takes command line arguments that will let you select digital or analog style, and 12 or 24 hour time. If you choose a digital display, you can also force the clock into the title bar area, out of the way.

You'll be able to change your Mouse pointer into a digital clock pointer with the ClockPtr command.

The Copy command can now retain comments (filenotes), creation dates and the protection bits of the original file. DiskDoctor now works somewhat better and first checks to see if it has enough memory to do its job.

A version of Microsmith's FastFonts text display speed-up program is now included. GraphicDump now takes command line arguments allowing a full size dump of the screen. Install now checks for valid boot blocks and clears memory before building a boot block. This change is a direct result of the virus programs and is intended to defeat many of them.

Finally, a speech handler (L:Speak-Handler) should now be included. It must be mounted as a logical device before use (SPEAK:). Once mounted, you can copy files into it and get spoken output or use it as a capture buffer for a terminal program so that the incoming text is spoken.

Again, let me stress that this is neither a complete list of all the changes expected in v1.3, nor is it an official list of changes that will definitely be included. I hope it is accurate, but as of this writing, v1.3 has not been released to the public, so things may yet change.

If nothing else, look at the list as things that are under consideration by Commodore; even if they don't make it into v1.3, they may make it into v1.4. V1.4 and v1.3 isn't even out yet?? Yes, Commodore is working on a version 1.4 of Kickstart/Workbench. While version 1.3 concentrated on making the CLI environment more useful, version 1.4 will concentrate on making the Workbench more useful.

# ACCESS

## *The world of non-commercial software*

**by Steve Ahlstrom**

*As the primary Sysop on Compuserve's Amiga forums, and a veteran Amiga programmer, Steve Ahlstrom sees just about every notable public domain and shareware program that gets written, and in this column he tells you about the ones that have recently caught his eye. If you have some information you'd like to share through this column, you can leave a message to Steve on CompuServe (his PPN is 76703,2006), or send the info to us and we'll pass it on.*

Telecommunications is very a popular use for personal computers. For some people, telecommunications is the *primary* function; whether it's for talking with commercial services, local BBSes, or working from home and calling the mainframe at work. In fact, I'm travelling across country as I write this column. When it's completed, I'll upload it to the Amiga Vendor Support Forum (a part of AmigaForum on CompuServe) where *TransAmi* maintains a customer support section.

There are various commercial, public domain, and shareware telecommunications programs available for the Amiga. Some are specialized terminal emulators (Handshake is a very fine VT-100 emulator), while most are generalized programs intended for communicating with commercial services and BBSes. In terms of features, they range from the minimal to the fully loaded. Many novice Amiga users believe that Amiga communications programs are not as 'full featured' as other programs they have used on other machines. True, but the reason for this, overlooked by some newcomers, is *multitasking*.

A recent Amiga convert stated that the program on his Atari ST was more powerful than what he could find on the Amiga. *On the surface*, this may be true. His program contained: a standalone text editor (with cut and paste), call logging for commercial services, extensive phone directory, and so on. He failed to realize that on a single-tasking computer those bells and whistles are almost a necessity. On the Amiga, they're the icing on the cake *at times:* but more often, they're a nuisance. As an example: the editor I use (uEdit by Rick Stiles) is perfect for me. I don't want to have to learn to use another editor; nor do I want to be forced to use an editor which, in comparison to uEdit, is lacking in features. I don't *want* a built-in text editor in my communications software. If it's there, it's just a waste of RAM space.

It's better to call up your text editor of choice from another CLI and load it with the review buffer from your communications program (if your program has a review buffer). That way, you've got your communications program running and your preferred editor! Other Amiga programs allow cut and paste without the need for adding that 'code bloat' to all programs. I use SnipIt by Scott Evernden. It allows you to copy text from any CON: window and paste it to any window expecting keyboard input. It's a very small, 'always resident' program. With the right combination of software, *there is no need* for the extra coding which is so desirable in communications software written for other machines. The same is true for other 'features'. The thing other computers lack is the multitasking.

**Sysop's choice**

My choice for an Amiga communications program is Access! by Keith Young. It's gone through many revisions with useful feature additions and ample sprays from the can of Raid. At the time of this writing, Access! is up to version 1.3a. (Note: There were two versions of 1.3; a straight 1.3 and a 1.3a. They are mostly identical except for one thing - a bug. If you have 1.3, there is a bug in non-windowed xmodem protocol. Replace that with 1.3a.)

Access! does not do terminal emulation (other than ANSI). It was intended to be an all-purpose generalized telecommunications program. I think it succeeds very well.

Access! is the only communications program that really takes advantage of the Amiga. It's colorful and loaded with gadgets, yet it doesn't tie you to the mouse (for those of you that find the use of the mouse to be less than second nature).

Access! lives in its own screen and can be run in four, eight, or 16 colour mode. You have full control of text and background colours. The colours of the gadgets are 'hardwired' as to which colour register they use, but all of the colour registers are user configurable.

Access! was written with multitasking in mind. It is a large program, but if you have sufficient RAM (one meg is recommended if you plan to do any serious multitasking), I've found

nothing that Access! interferes with. The sole exception is when running another copy of Access!. In that case, you will get serial port conflicts and strange things will happen. The program even allows you to set its priority, so you may want to increase it or decrease it depending on the requirements of other programs you may be running. The priority setting is just one of the many user configurable items which can be saved to a configuration file that will define the default environment the next time you run the program.

If you want to run in interlace mode, Access! will oblige quite easily. I normally run it interlace, using Topaz 11 which I find easier to read with less flicker on my monitor. While Access! does not support fonts directly, it works well using Dave Haynie's SetFont 2.0. If you use interlace, there is a feature which I find handy. Click on the 'split' gadget and the 640*400 screen becomes 640*200. You then have direct access to the Workbench screen behind it without having to flip screens. The flip option is only available when you're in interlace mode since it halves the number of text lines available to you. On the normal 640*200 screen, the amount of text you could display on screen would be severely restricted.

Access! offers so much in useful features that to describe them all would exceed the space limitations of this column. I'll list most of them. As I mentioned earlier, I'm on the road, and unfortunately using an MS-DOS laptop at the moment, so if I leave out some features, I'll apologize to you now (and to Keith!).

Access! contains a 60-entry phone book. The phone book gives you ample room for the name, baud rate, parity settings, and comments. You can load in alternative phone directories.

Access! supports xmodem, wxmodem, CompuServe QB protocol (if you use CompuServe, QB is the way to go. It's the fastest of the file transfer protocols available on any of the commercial services), and ASCII transfers.

In addition to the fully configurable colours, you can set Access! up for any combination of parity/bit settings. The program has two clocks, a real-time clock and an online clock. It has an editable chat window for online conferencing and a non-editable review buffer. The review buffer allows you to select the number of lines the buffer will contain, from 50 to 500. The review buffer can consume quite a bit of memory. Figure 500 lines times 80 characters per line and you'll see this can eat up memory, but it's a handy feature to have. I have mine set to the max, 500. (Gee, what did you say in conference five minutes ago? Lemme go see.)

Of course it also has macros, a limited script language (although I don't believe that's documented yet), and even more features than I can remember right now (oh, for a laptop Amiga!). Practically every user-configurable item can be saved to a config file.

Access! isn't for everyone, especially those needing specific terminal emulations, and it's not the 'be-all and end-all' for me, but it's a very fine implementation of an all-purpose Amiga communications program.

Of all the communications programs now available, including the commercial contenders, I believe that Access! is the best of the bunch. Keith Young is very receptive to user comments and complaints, so I'm sure as time passes it will become even more powerful and bug free. Access! is shareware and worth every cent!

**Making your Amiga fly**

Co-Pilot by Jack Radigan is a very handy utility to have. It allows you to define 276 macros. These macros can be command lines for the CLI or they can be mouse move/keypress sequences.

Co-Pilot includes a real-time clock, a screen blanker, day of week indicator, and a lot more. It's one of those programs that I run from my startup-sequence and forget about.

The ability to define mouse movement and keystroke sequences as macros has powerful implications. With these features you can set up fully automated demonstration sequences. With a single macro invocation you can: bring up Dpaint, create a picture, save it to disk, exit Dpaint, run a slideshow program including the picture you just created. Text creation and data entry can be greatly accelerated by the use of macros. These are just very simple examples of what can be done. The possibilities are almost endless.

Co-Pilot will also cycle through screens. How many times have you had the Workbench screen up and a couple of other programs that run in their own screens (say, Access! and uEdit)? The operating system left Amiga-M/N sequence only flips between the top two screens. To get to the third screen takes some mouse/keyboard gymnastics. With the screen flipper feature of Co-Pilot, the current top screen is pushed to the rear. Regardless of how many screens you have running, you can always (easily) get to the screen you want to work on.

A similar feature is available for windows. Co-Pilot builds and maintains a window list. A simple keystroke will push the current window to the back and bring the next window in the list to the front. Another keystroke will activate windows while not bringing them to the front. These features should have been part of the operating system but aren't.

The authors, Keith Young and Jack Radigan, are both active on the commercial services (CompuServe's AmigaForum and GEnie's Amiga Roundtable). Unlike many shareware authors, they are very interested in your opinions on their work and your suggestions for future features. If you'd like more information on these programs, feel free to contact the authors directly.

I'm going to cut it short this time. Until next time, keep those super freely-distributable programs coming!

# ViewPort

## *Let the Amiga make it on its own*

**by Larry Phillips**

*Larry Phillips is an Amiga hardware and software hacker from Vancouver, and a Sysop on CompuServe's Amiga forums.*

I was asked recently to sign a petition. Well, it wasn't as if the person asked me directly, but I was asked as part of a larger group. The petition was for the purpose of convincing Ashton Tate to port dBase III to the Amiga. I didn't sign it, though it would have been easy for me to do, as the petition was in electronic form and would have required nothing more than sending electronic mail to the person soliciting signatures. In the discussion that ensued, I voiced my opinion that not everyone cares about Ashton Tate or their programs, that porting programs from the IBM world might be counterproductive in the long run, that there just might be more to life than worrying about how many data base managers there are for a particular brand of computer, and that I really don't want to sell the Amiga on the basis of how many IBM programs it will run. I have an A2000, but not a Bridge Card, nor will I ever have a Bridge Card, which will serve to show you how I feel about the value of empty space as compared to an MS-DOS clone.

The argument urging me to sign was hard to resist. If enough people signed, the theory went, Ashton Tate might just port dBase to the Amiga. It was hoped that this action would cause the entire business community to flock to their local Amiga dealers and seriously deplete inventories, making CBM rich and Amiga users in general very happy that their favourite machine had at last 'made it'. I did resist though, and feel that I did the right thing.

The first reason I didn't sign, though it has nothing to do with the theory above, is that I have absolutely no use for a spreadsheet, a database manager, accounting packages, or any other business-oriented program. I might someday find a use for one or more business programs, but right now they would moulder on my shelf. Since I don't need any of these programs, I have little interest in seeing them ported. There are already a lot more Amiga business programs than I would ever require.

The second reason concerns the theory put forward by the author of the petition. I have seen many programs that were either direct ports from the IBM world or were written in a style indistinguishable from a direct port, and frankly, though they may do the job they are supposed to do, they just don't 'feel' right for the Amiga.

I can understand the argument that there are potential customers who need to be able to run a particular piece of software at home, and would rather do it on an Amiga. I can understand the argument that there will be sales directly attributable to the availability of said software on the Amiga. I can envisage die-hard IBMers just looking for an excuse to dump their clones and buy a machine like the Amiga, and finding it in the form of a Lotus or dBase clone. I can see all that, yet I didn't sign the petition. To say it simply, I just don't care.

There is something distasteful in the idea that there are people who can only be convinced to consider an Amiga if we have IBM-style programs running on it. To take programs like Lotus and dBase, with their drab, IBMish, text-oriented user interfaces, and port them unaltered to the Amiga, seems somehow counter to the very ideals of those who enjoy the Amiga for what it is, and doubly so for those who can enjoy the Amiga for what it can be.

There is also something distasteful about using a ported program to entice business users to buy the machine. It smacks of trickery. "We'll just get this unsuspecting business type to buy one, then we'll wow him with better programs later." One of the problems with this approach is that those 'better programs' might never be written. Who wants to compete with Ashton Tate or Lotus? How many authors would say "Well, there goes the idea of writing a spreadsheet. We already have the top dog, the industry standard."

The Amiga is unique in today's world of small computers. It is poised between the big markets, and between markets that will be big in the future. It is neither business computer nor engineering workstation - not, at least, according to those too blind to see its current capabilities and future potential. The Amiga has already found many niches to occupy: video production and post-production; home video; colour desktop publishing; music; arts, and more. Do we really want it to attempt to enter the cutthroat business market in a full frontal attack, with its ported IBM software, competing against sterile, boring machines that are already established as 'the right stuff' by those with all the vision of a box full of hammers?

I would rather see the Amiga 'make it on its own merits'. I would rather show someone what it can do; light a fire in his imagination by showing him how far the machine has come in

three short years, and giving him a glimpse of what is to come. I would rather make him want the Amiga for what he can see it doing for him with programs that are better and more beautiful, with multitasking, with cooperating programs that can be mixed and matched, rather than having to be purchased as an 'integrated package'. I want the business users who have Amigas to scream loud and long at any company who would dare to put out a clone of an IBM program for it.

Yes, the IBM programs running on the Amiga will generate sales in the business community, many more than will be made by trying to convince unimaginative men in three-piece suits of the beauty and power of the Amiga, but think of what the long term results will be. The businessman who buys for the IBMishness of the Amiga as he sees it will be buying with all the eagerness he would exhibit when he buys a typewriter or a box of paper clips. The one who buys the Amiga because he sees its power and beauty is investing in a dream. He is buying because he is an enthusiast. He becomes a zealot, preaching the gospel of multitasking, user friendliness, colour and sound, to anyone who'll listen. When he hits upon just the right thing to show a colleague, he will convert him too, and the chain goes on.

No, the business world can keep going to IBM and the clone makers if that's all it takes to satisfy them. They can keep on reading Jerry Pournelle's columns in *Byte* or Steve Gibson's columns in *InfoWorld*, happy as a cow on the way to slaughter, reinforced in their belief that they have made all the right choices, studiously putting their blinders on every morning to avoid seeing anything that even hints of imagination.

The petition? Well, you can sign it if you want to. I would never think to condemn you for wanting Lotus 123, and simply want to point out that there is another side to it, that the silver cloud might hide a black hole that could suck the Amiga down to the level of IBM clone.

To my way of thinking, if you really want to help the Amiga make it in the business market, you will find someone who has the power to make buying decisions, sit him down in front of an Amiga, and blow his socks off. Show him everything. Show him productivity the likes of which he can't get with a single-tasking computer, and can't get at all on any other machine at five times the Amiga's price. Show him graphics, sound, business programs, video applications, anything that you think might turn him on to the machine. If you have to, tell him about the Bridge Card. Tell him how much it will cost. Make that light come on in his mind.

Be prepared for some disappointments, because it won't always work. It's often impossible to convince the man who thinks that IBM is the ultimate in high technology that there are other, better machines. If you can't convince him, don't worry about it too much. Find someone else to show. Sooner or later you're bound to find one with vision. And if you never find one, at least you can rest assured that *you* have the vision to own a machine that has more going for it than the ability to run dBase or Lotus.


# We're looking for some far out thinkers!

*"We're far out and very bright!"*

You don't have to be from outer space. You just have to be a unique computer programmer.

If you have an Amiga program near completion that you think is a winner, you should contact Gold Disk right away. We're looking for top quality software to market. Working with us could be the most rewarding move of your life.

Write or call us for our free developer's kit, "Hitch your wagon to a star."

GOLD DISK
P.O. Box 789, Streetsville, Mississauga
Ontario, Canada L5M 2C2
Telephone: (416) 828-0913

# BreakPoint Part Three

## *Third in a series on debugging*

**by Victor A. Wagner**

*Vic Wagner started his association with computers in 1965 while in the US Air Force working in a group studying digital flight simulation. Since becoming a civilian in '66, he has been working mostly with minicomputer manufacturers in the real-time systems software area. Weekdays from 8 to 5 find Vic answering the technical support line for Computer Automation Inc., and maintaining three real-time software systems. Evenings and weekends he spends most of his waking hours talking to Taarna (his Amiga 1000), writing programs, and spending time on CompuServe's Amiga forums (76046,3004). Vic is a recognized authority on Metadigm Inc.'s well-known MetaScope debugging program; he is also President of the Amiga Friends users' group.*

This month we will examine the effects of several innovations in computers and how they affected *Testing*. As we shall see, not all the changes were positive. While most would say that we have taken giant strides forward in software and testing, I'm not sure that there haven't been a few giant steps backward as well.

Let's take a look at this technology we've wrought. First let's see what the CRT did for us. Now here I'm talking about the less expensive character (not graphic) CRT. CRT displays had existed for some time, but their cost was generally quite high. Around 1965 (I think) Raytheon invented the raster scan character display. This allowed inexpensive video systems to be used to display characters. TV costs were coming down, and raster scan is what a TV uses to display its picture. The technology for doing such monitors was well known. (For the real techies out there, you could use the horizontal oscillator to help generate the high voltage necessary for the CRT, instead of having to come up with some other method.)

At any rate, CRTs became common. They were very fast devices, compared to the old console I/O writers (which we will call 'TTYs', because TeleType Corporation had made inexpensive keyboard/printers that were in wide use by this time). At any rate, the standard TTY printed about 10 characters per second, while an 'average' CRT would output around 1000 characters per second.

Wow! We can output stuff a hundred times as fast!

*Oh-oh, wait a minute, we can't read that fast.*

That's okay, we can always read it later at our convenience!

*Oh-oh, there isn't any paper in this thing, where'd all the data go? We used to be able to pick up the paper off the floor when we'd dumped a lot of data with the debugger. Not any more.*

So, the CRT had a benefit (faster output) and some drawbacks (only 24 lines visible at a time). This caused some enhancements in debuggers, and certainly changed the way we looked at memory. If nothing else, we certainly never asked to dump a thousand words of memory on a CRT... well, not intentionally anyway. It also had no 'logging' capability like the ubiquitous TTY, so we couldn't easily go back and see what we had done.

Time to rethink some old habits. No more dumping a couple of hundred locations at a time (we could safely go get a cup of coffee while dumping *that* much memory), highlighting the relevant places, adding it to the pile by our elbow and carrying on. Okay, so we put some limits into our debugger so we don't accidentally push all of the important information into the 'bit bucket'. If you were lucky, the debugger would have an option to print such dumps on a printer. If you were *very* lucky, the line printer would be in the same room.

Pretty soon, we had all sorts of papers lying around, marked with highlighter. Finding the appropriate piece could be difficult. Time to make the debugger smarter (or at least simpler to use). A couple of features were generally added:

1) Some special user-definable symbols were added. These symbols were generally of the form R0-R9, or G0-GF, or similar. They could be assigned addresses, and were used as 'base' registers for the debugger. Now when we wanted to look at our input buffer (to which we had 'assigned' the base register R0) we could just say 'examine' R0, instead of 'examine' 123F (our input buffer's absolute address). This was much more of a timesaver than it at first appears. We no longer had to have all

those pieces of paper laying around. By simply assigning a base to the interesting places in the program, we could look at any area at whim. It's usually a lot faster to type 'LR3' (list from R3) than to locate the relevant stack of paper.

2) Debuggers began to allow expressions in their input. This allowed us to say I123+456 to look at location 579 (the letter 'I' is a mnemonic for 'Inspect'; I'll be using it in the sample commands from now on). If we were lucky, I123R0 would be equivalent to I123+R0. If we were very lucky, the debugger would have special symbols for the registers that would be legal in expressions (like say X1, X2 and X3 for the index registers). This gave us the ability to say IX3 and see the memory that X3 was pointing at. Or, if X3 were pointing at some structure in which the 5th word were important, we could enter I5X3.

Things are beginning to really look up. Now, if only we didn't have to keep these really strange hours... Which brings us to the next advance:

Time-sharing computers. As we all know, computers can only do one thing at a time (broadly speaking). Some rather bright people (I don't remember where) discovered that a computer spent a lot of time waiting for I/O devices. They noted that when the interface to the editor was a CRT and a human, a *lot* of time was spent waiting for the human to type characters. Even more time is spent waiting for a human to input data to a debugger. These people managed to write systems that allowed many users to have the illusion of having sole use of a machine. You could actually sit at your CRT and believe that you had the exclusive use of the machine. Well, the compiles and assemblies took a little longer, but you spent most of your time thinking, editing, or debugging. Besides, when you wanted to assemble, you could always go get a cup of coffee while the machine was busy working on your program.

*People managed to write systems that allowed many users to have the illusion of having sole use of a machine...*

We could actually debug during normal working hours. No more 3 a.m. debugging sessions. Well, not all was bliss - we had *meetings* during normal hours. As a matter of fact, most programmers discovered quickly that the 'normal business hours' were the least productive. So there still were 3 a.m. debugging sessions. We did learn how to work together, how to split up large projects, how to share data and files. We learned, most important, how to always have it appear to be the other guy's fault; so *we* didn't have to stay late. (Actually, that's a base canard - almost all the teams I was associated with were really more interested in getting the job done than in determining blame.)

We had all heard the sermons: We were going to get rid of the paper money, funds would be transferred electronically. Paper mail would be a thing of the past. Everything would be stored electronically.

We even believed it. We started to look at programs through a window of 24 lines by 80 characters. Compile/link times were short enough (less time than to go get the listings) that we would do another run on the computer rather than walk to where the printouts were stored. Now don't get me wrong, I've nothing against rapid prototyping, but I'm talking here about production programs being created in this way. Well, it did push some of the arts of testing in the right direction.

We now had debuggers that would read files that told them what to do. We needed the ability to do some symbolic entries, because we were never sure where our program would be loaded. We didn't want to have to walk all the way over to the printer to get the link map, then edit the debug file and correct the addresses. Computers were designed to reduce the drudgery and to do the repetitive tasks. Of course, we occasionally needed to produce a lot of output, and the printer was the only place to put it, so we would generally print the listing at the same time (we only had to walk to the printer once). Then we would sit down and pore over the debug output and the listing, and plan our next attack.

(Just so you'll know where I'm coming from, I'd better tell you that the rest of this article will be based solely on my experiences. Since 1971 I have been involved exclusively with mini-computers in the real-time embedded systems area. My awareness of things going on in the 'main-frame' world is woefully lacking. I hear of fantastic debuggers on the big machines. I hear of Source Code Control Systems (which keep track of what you and all your teammates are doing to the program(s)). I hear of programming groups with hundreds (well, scores, anyway) of programmers. The largest group I've been associated with was a software department of 24 people, including managers and secretaries. My perceptions of where software testing have been is necessarily coloured by this provincialism over the last seventeen (has it really been that long?) years. This time has been spent at only two companies, with my tenure at the second accounting for nearly all of it. Okay, so it's back to 1969 and into the mini world.)

An interesting thing about going to work for a mini-computer company was that there were (for the first time in my career) computers plural. After all, we make and sell them, so there are always some lying around. Actually we have a rather well stocked 'development' lab. There are only four of us in software, including the manager, and we have a total of four computers (no, not one for each; this company makes two families of computers, and we have two of each). We do have keypunches, card readers, TTYs, paper-tape punches (and readers), and even a line printer (which is shared among the four machines). There is practically no software for these machines yet, so we are developing the operating systems. My job is to write the assembler for the 8-bit machine.

Have you ever considered how you go about writing an assembler for a machine that doesn't have one? Well, I didn't have that problem, it's just that the new machine had many new features and they wanted the assembler "totally redone" for the latest in the family. This meant that I had to 'write' the program twice. Once in the old assembler, to get a minimal subset working, then recode in the new assembler and bootstrap. This was an interesting experience. We also had to write a new debugger because we were in the process of changing from octal to hex representations. Talk about confusing. We had just about given up on ever being able to efficiently do things (we did have a 12.5 ips mag tape system) when a new 'inexpensive' solution presented itself. And a new era of testing problems/solutions appeared.

Disks and disk-based operating systems... One thing is certain, the advent of inexpensive disk drives changed the face of computing. Now everyone could afford to have a disk-pack of their own. In the early 1970s, removable 5 megabyte 14-inch disk packs cost about a hundred dollars US. In the minicomputer area (micros still hadn't been invented), a popular drive consisted of 5 meg fixed and 5 meg removable. The price of the computer system had dropped to one year's salary for a programmer, so machines were beginning to proliferate. Not that each programmer had a machine (that comes much later), but an installation is likely to have more than one machine for staff use. Disk based operating systems spring up and now we can assemble/compile our source at blinding speeds (actually, even the card-reader technology had become quite advanced in those years... I have worked with card readers which could read 1000 cards a minute). We could keep our source on this 14-inch diameter, 2-inch thick portable disk. Hmmm, it isn't quite as easy to edit this disk as it was the cards though. It's difficult to grab a couple of source lines off the disk and replace them with a few new ones.

The fact that humans cannot directly manipulate the data on disks brings a new program into existence: the Editor. With this marvellous program we can add, delete, and replace source lines in our programs, just as we can with a deck of cards. Of course, the early editors required that we 'sort' the changes so that they could be done in a single pass, but no matter, at least we weren't hauling thousands of cards around.

*The fact that humans cannot directly manipulate the data on disks brings a new program into existence: the Editor...*

We still had our keypunches though. We weren't rich enough to let someone use a computer just to enter programs, or for that matter, to waste an entire computer just editing a source file. We would keypunch the directions to the editor to fix our source file. Take the cards with a few control cards to the machine. Run the edit. Run the assembler/compiler. Do the link. Grab the paper-tape loadable (yes, paper-tape; card punches were very expensive) and go off to one of the debugging machines. Oh yes, and don't forget to grab the listing and the link map off the line-printer.

I guess the big advantage here really wasn't so much the disks and disk operating systems as the availability of more than one machine. We could afford to have a 'development' machine as well as a debugging machine (or several of them). We still had to schedule the machines, but the scheduling was much more reasonable (not all debugging was done at 3 a.m.). Why, if you were moderately careful, you could actually get several edit/compile/debug sessions done in a day.

An interesting problem arose as the price of disks came down. It became very easy for each programmer to have a disk. This caused, among other things, duplicates of some files to exist. It was difficult, at best, to ensure that each programmer had the latest copy of the global files. One of the most frustrating things is to spend a substantial amount of time chasing down a problem that has already been fixed. This problem of multiple copies of files won't be fixed until we get to LANs (Local Area Networks), a topic that is outside the scope of this article (mostly because my experience with debugging on them is limited to a single project). To continue however:

Several things then happened at nearly the same time.

- Machine costs came down so that we could afford to have machines on which people could edit their source (coincidentally, card readers seemed to be going out of style in the mini-world).
- Editors became more sophisticated. You could actually 'page' through your file and make changes in any order you wished.
- High Level Languages began to become popular; specifically, the block structured languages like Algol, Pascal, BCPL and C).

I believe that a substantial change came over the programming world during this time. I believe that we went from being a society of thinkers and planners to a society of doers. This may have been true of American society as a whole, but it appeared (to me) to be much more severe in the computing industry. In my opinion, the next step exacerbated the situation.

Personal computers began to thrust themselves upon the awareness of a rather unsuspecting world. Of course, we were all aware of microcomputers and, unless we were actually building one at home, we looked down upon them as curious toys. They were for play. They only had 16K of memory. They had no software. You couldn't even see what they were doing with a control panel. Why would I want one? I've got ten times the memory (and power) in just *one* of the machines here at the office. Well, perhaps if I wanted to learn about some fundamentals of hardware, but you couldn't do real software on them.

There were insidious plans afoot, however. Well-meaning individuals were actually trying to put together operating systems for these micros. Assemblers were written that actually ran on the micros. These machines were too small and inexpensive to support expensive peripherals like card readers and punches or line-printers, so other means had to be found to work with them. That something was the floppy disk. These disks didn't have a lot of capacity (128K) but it was enough. CP/M came into existence. It was a marvel for its time. It supported a file system and a mechanism for running programs.

Editors were quickly developed, assemblers, even some compilers. Some programs for people who weren't even programmers were developed (WordStar™ comes to mind). Several companies were packaging micros and writing systems and selling them to businesses. These machines were inexpensive by any other standards. Small businesses could afford them. More and more people were getting involved in the microcomputer business.

And business it was. Sure, individuals owned these machines, but they were 'hackers' (then an honourable term), hardware fanatics, and I suppose a bunch of small start-up companies trying to build business systems. The only debugging program of note (well, that I noticed) to come out of all this was a debugger by Digital Research with the cute name of DDT (which they claimed was an acronym for Dynamic Debugging Tool). Well the Intel 8080 and the Zilog Z80 were actually beginning to sell, and sell in large quantities. Of course most were embedded in boxes somewhere, controlling widgets, but they were off and running. Then the bomb dropped:

*International Business Machines announced the PC... Most pundits of the day (myself included) predicted that this was a **big** mistake...*

Adam Osborne announced a machine that was portable (well, 27 pounds). It was complete with *two* disk drives (100K each) and a monitor (5-inch screen). It sold for around $2500 US. Included with this system would be around $400 worth of software including WordStar™ (a 'word processor') and SuperCalc™ (a 'spreadsheet'). Now here was a really low-cost machine, and it already had software to go with it. Ordinary people could use this machine. You didn't need to be a 'techie' to operate it. SuperCalc and WordStar made the machine useful to regular people. It even had CBASIC and Microsoft BASIC for those who wanted to write their own programs. Osborne I's were sold in bunches. Programs that ran on all CP/M machines were written in bunches. Personal computing had arrived. My wife and I even bought one (after lots of soul searching and justifying, that is). Then the other bomb dropped:

International Business Machines announced the PC. IBM?? You mean the people who make *real* computers? Yup, them. Most pundits of the day (myself included) predicted that this was a *big* mistake for IBM. What did they know about retailing? When had they *ever* been able to deal with customers without that condescending sneer? Regular people wouldn't tolerate that kind of behaviour.

Well, all the pundits (what the heck *is* a pundit?) were wrong. Business flocked around the PC. All those middle managers who had *had* it with the MIS (Management Information Systems) departments were getting their own machines to do 'word processing' and 'data analysis'. The business of computers and writing software for them resembled the Bikini Atoll tests in the early 1950s. Lots and lots of software was written and (sometimes hard to believe) tested.

Debuggers remained broadly the same, but a new wrinkle appeared. Disassemblers were now embedded in the debuggers. You could ask to see memory either as data (like all the dumps before) or you could view it as instructions. This was especially important on some of the micros as it is extremely difficult to decipher by hand. Now you can view the instructions the computer is going to execute. You don't even have to bring the listing of the program (if it's in assembler). You *do* have to write down any changes you make, so you can edit them later, but we are definitely on the right track. Now if I didn't have to write the changes down...

We now arrive at the Amiga. The first multitasking (out of the box) personal computer you can buy. I'm sure you've all at least played with the multitasking features of your Amiga. You know how simple it is to push the current program's window to the back, perhaps open another CLI or open UEdit's tiny window, and edit some source without exiting the program you were using. If you're like me (I'm always convinced that the bug I've just found is the last one), you'll start the compile/assemble/link process as soon as you've finished the edit. This way, when you've finished verifying that the program is indeed correct, the finished product will have already been made. Boy, this testing/fixing couldn't be better... unless...

*I wish the output from the program didn't destroy the debugging information.*

AmigaDOS and Intuition to the rescue again! If the debugger opens its own window, the program output will appear one place, and the debugging information somewhere else! Gee, when my program prints out its instructions, I can still see the debugging information. If the debugger can open *more* than one window, I can see the registers in one, a buffer or two in another or two, some variables in still another. Wow! Heaven!!

Next month we'll take a look at four widely available debuggers for the Amiga. We will show how to use each on a simple program, and try to show off the features of each. The debuggers we will look at are: *db*, *sdb*, LDebug and MetaScope.

# 3-D Plotting Routines for AmigaBasic

## *Display and animate 3-D graphics*

**by Anthony Bryant**

*Anthony Bryant is a graduate of the University of Manitoba. He is a computer engineer working with remote control systems and robotics, and programs the Amiga in his spare time.*

AmigaBasic has such a rich storehouse of commands that its facility for creating new commands is sometimes overlooked. Subprograms are the way to expand the command language with application-specific new functions.

My collection of subprograms, PLOT 3D, is a set of 3D drawing tools, capable of plotting points in 3D coordinate space, as defined by parameters passed to each function. The subprogram set is easily appended to any application program, without concern for redefinition of variables.

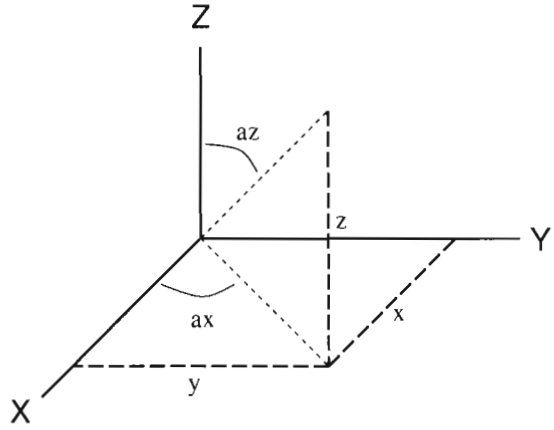
As an example of its use, Listing 1 demonstrates four views of a sphere quadrant as seen from different observation angles. (Don't forget to append the subprograms, PLOT 3D.)

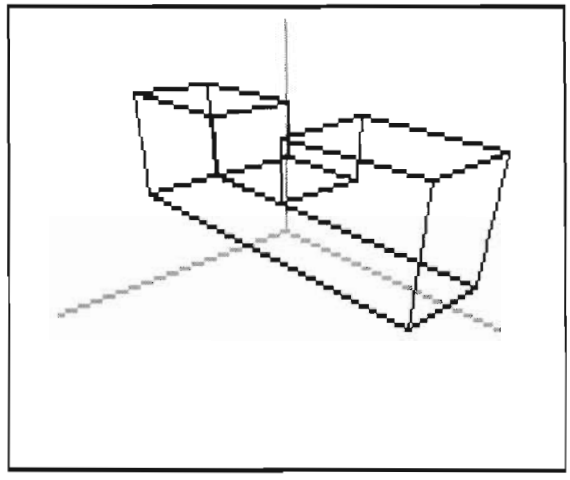
Listing 2 shows how to rotate a shape (as specified in the DATA statements) about the Z-axis. By changing the observation angle, ax, each time the subroutine is called, different views are 'recorded' in a dimensioned array and 'playback' animation is achieved using a GET/PUT loop to flip through the 30 frames quite quickly. Since the dimensioned arrays take up so much space, this program needs as much memory in the Amiga as is possible (for a 512K system). It's a squeeze!

Using the present set of subprograms, PLOT 3D, as a foundation, you may wish to expand further with ever-new graphics commands.

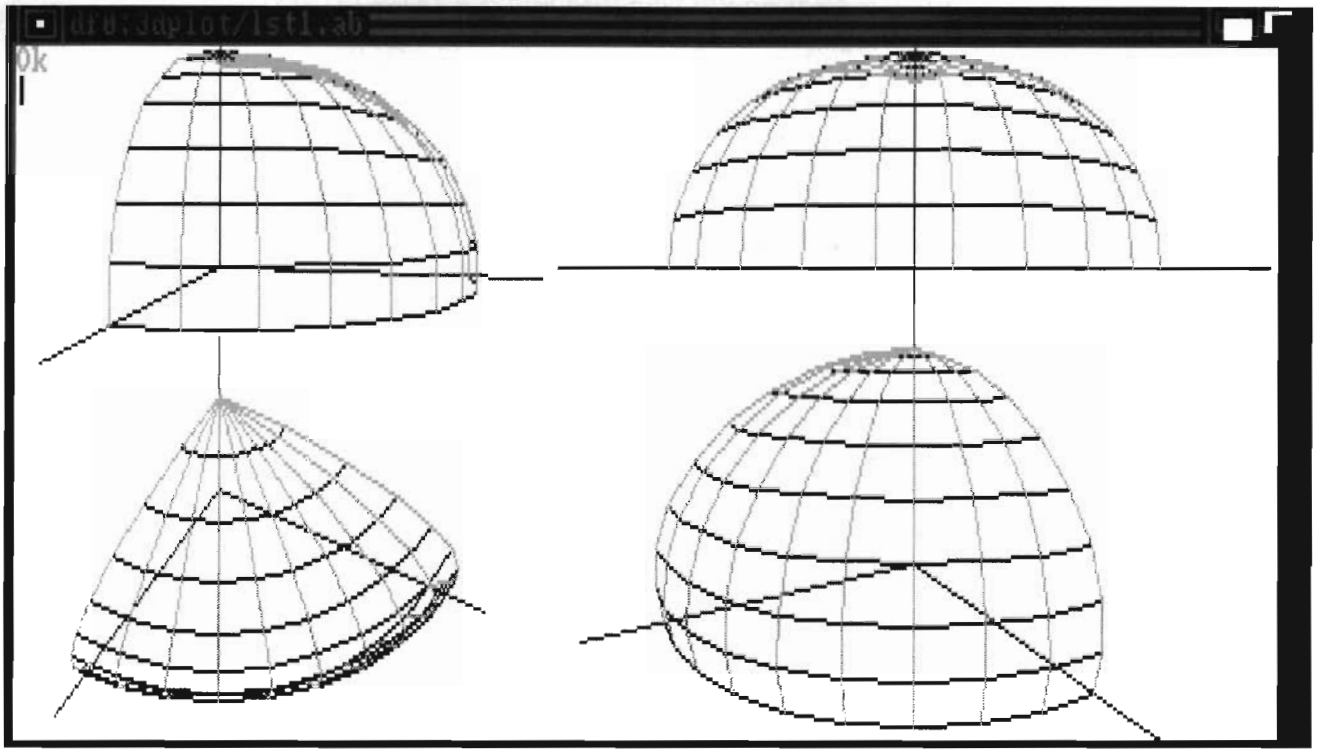
These are the subprograms in the PLOT 3D collection: see the program listings for usage examples:

- **dMoveto (x, y, z)** - Move to a given coordinate and plot a single pixel using the current COLOR.
- **dDrawto (x, y, z)** - Draw from previous to given coordinate
- **dAngle (ax, az)** - Set observation angle in degrees (see diagram)
- **dParams (x, y, sc, di)** - x and y set the screen coordinates of the origin; sc is a scaling factor; and di is an additonal scaling factor for the z axis. (100 and 2.2 are good values to use for sc and di).
- **dAxes (xL, yL, zL)** - Draw axes of given size



*The rotating figure created by listing 2, caught in mid-turn. Each frame in the rotation sequence is calculated, plotted and recorded, then the recorded frames are played back in real time.*




*The display created by Listing 1, which plots a sphere quadrant using the 3-D plot routines and displays it from different observation angles.*

```
' Listing 1.
' for 512k  AmigaBASIC

' demonstration program for PLOT 3D
' change observation angles, each view
' using new graphics commands in subprograms

FourViews:
  COLOR 1,2: CLS
  dParams 100!, 140!, 100!, 2.2
  dAngle 15!, 80!
  dAxes  75!, 75!, 75!
  GOSUB PlotQSphere
  dParams 100!, 80!, 100!, 2.2
  dAngle 30!, 15!
  dAxes  75!, 75!, 75!
  GOSUB PlotQSphere
  dParams 440!, 140!, 100!, 2.2
  dAngle 45!, 90!
  dAxes  75!, 75!, 75!
  GOSUB PlotQSphere
  dParams 440!, 60!, 100!, 2.2
  dAngle 60!, 60!
  dAxes  75!, 75!, 75!
  GOSUB PlotQSphere
END

  'the subroutine

PlotQSphere:
  COLOR 1: r=60: rad=3.14159/180
  FOR theta=0 TO 90 STEP 10
    z=r*COS(theta*rad): r1=r*SIN(theta*rad)
    x=r1:y=0: dMoveto x, y, z
    FOR beta=0 TO 90 STEP 10
      x=r1*COS(beta*rad): y=r1*SIN(beta*rad)
      dDrawto x,y,z
    NEXT beta
  NEXT theta
  COLOR 3
  FOR theta=0 TO 90 STEP 10
    x=0: y=0: z=r: dMoveto x,y,z
    FOR beta=0 TO 90 STEP 10
      z=r*COS(beta*rad)
      x=r*SIN(beta*rad)*COS(theta*rad)
      y=r*SIN(beta*rad)*SIN(theta*rad)
      dDrawto x,y,z
    NEXT beta
  NEXT theta
RETURN

 ************************************

' Listing 2.
' for 512k  AmigaBASIC

' rotation of a shape about the Z-axis
' by changing the observation angle, ax
' using new graphics commands in the subprograms

Rotation:
  CLEAR  ,186000& 'make room for arrays
```

```
  COLOR 1,2: CLS
  xo=140: yo=100: sc=100: di=2.2
  dParams xo,yo,sc,di
  xl=xo-110: xr=xo+110: yt=yo-50: yb=yo+50
  LINE (xl,yt)-(xr,yb), 3, b
  ax=0: az=60: GOSUB PlotFigure
  n=2830  'array size
  f=30    'frame number
  DIM  array%(n,f), blank%(n)
 Record:
  LOCATE 5,8: PRINT "Recording ..."
  LINE (xl+1,yt+1)-(xr-1,yb-1), 2, bf
  GET (xl,yt)-(xr,yb), blank%
  FOR f=1 TO 30
    GOSUB PlotFigure : ax=ax+6
    GET (xl,yt)-(xr,yb), array%(0,f)
    PUT (xl,yt), blank%, PSET
  NEXT f

 Playback:
  LOCATE 5,8: PRINT "Playback     "
  WHILE 1  'forever
  FOR f=1 TO 30
    PUT (xl,yt), array%(0,f), PSET
  NEXT f
  WEND
END

   'the subroutine

PlotFigure:
  dAngle ax, az
  dAxes  50!, 50!, 50!
  COLOR 1
  RESTORE Figuredata
  FOR pnt=1 TO 33
    READ x, y, z, i
    IF i=0 THEN
      dMoveto x, y, z
    ELSE
      dDrawto x, y, z
    END IF
  NEXT pnt
RETURN

  'sample shape data

Figuredata:
 '       x,  y,  z, i (i=0 for moveto)
  DATA -10,-10, 30, 0
  DATA -10,-40, 30, 1
  DATA -10,-40,  0, 1
  DATA -10, 40,  0, 1
  DATA -10, 40, 30, 1
  DATA  10, 40, 30, 1
  DATA  10, 40,  0, 1
  DATA  10,-40,  0, 1
  DATA  10,-40, 30, 1
  DATA  10,-10, 30, 1
  DATA -10,-40, 30, 0
  DATA  10,-40, 30, 1
  DATA -10,-40,  0, 0
  DATA  10,-40,  0, 1
  DATA -10, 40,  0, 0
  DATA  10, 40,  0, 1
  DATA  10, 10, 30, 0
  DATA  10, 40, 30, 1
  DATA -10, 10, 30, 0
  DATA -10, 40, 30, 1
  DATA -10,-10, 30, 0
  DATA  10,-10, 30, 1
  DATA  10,-10, 15, 1
  DATA -10,-10, 15, 1
  DATA -10, 10, 15, 1
  DATA -10, 10, 30, 1
  DATA  10, 10, 30, 1
  DATA  10, 10, 15, 1
  DATA  10,-10, 15, 1
  DATA  10, 10, 15, 0
  DATA -10, 10, 15, 1
  DATA -10,-10, 30, 0
  DATA -10,-10, 15, 1


 ************************************

    'the subprograms for 3-d plotting
PLOT3D:
SUB  dMoveto (x, y, z)  STATIC
  dTransform  x, y, z, sx, sy
  PSET (sx, sy)
END SUB

SUB  dDrawto  (x, y, z)  STATIC
  dTransform  x, y, z, sx, sy
  LINE - (sx, sy)
END SUB

'  transform a point to screen coords
SUB  dTransform  (x, y, z, sx, sy)  STATIC
SHARED  s1, c1, s2, c2, scx, scy, xo, yo
  xe=-x*s1 + y*c1
  ye=-x*c1*c2 - y*s1*c2 + z*s2
  ze=-x*s2*c1 - y*s2*s1 - z*c2+ 120
  sx=scx*xe/ze + xo
  sy=199 - (scy*ye/ze + yo)
END SUB

'  set up observation angles in degrees
SUB  dAngle  (ax, az)  STATIC
SHARED  s1, c1, s2, c2
  pi=3.14159: rad=pi/180
  s1=SIN(ax*rad): c1=COS(ax*rad)
  s2=SIN(az*rad): c2=COS(az*rad)
END SUB

'  set up screen parameters
SUB dParams  (x, y, sc, di)  STATIC
SHARED  xo, yo, scx, scy
xo=x: yo=y: scy=sc: scx=sc*di
END SUB

'  draw  3-d axes
SUB dAxes  (xl, yl, zl)  STATIC
  COLOR 0
  dMoveto 0!, 0!, 0!
  dDrawto 0!, 0!, zl
  dMoveto 0!, 0!, 0!
  dDrawto 0!, yl, 0!
  dMoveto 0!, 0!, 0!
  dDrawto xl, 0!, 0!
END SUB

 ************************************
```

# The TOP GUN of AMIGA ASSEMBLERS is:

# C.A.P.E.68k!

## The "Complete Assembler Programming Environment" from Inovatronics, Inc.!

Count on Inovatronics to get the most speed of development AND execution out of the AMA* assembler standard while putting in more useful control features than any other Amiga assembler. Features like these:

- Integrated assembler/editor with "assemble from buffer" and "stop on error"
- Code format options: EXEcutable OBJect, Binary, ALink format, etc.
- AREXX macro support for easy session automation
- Program size and number of modules unlimited
- Excellent code optimization (optional)
- Usage of pre-compiled includes, which net a 10x speed increase over standard includes used by other assemblers.

| Features: | C.A.P.E. 68k | MetaComCo | Devpac | Assempro | Manx |
|---|---|---|---|---|---|
| Integrated Editor/Assembler | Yes | No | Yes | Yes | No |
| Generates Standalone | Yes | No | Yes | Yes | No |
| Generates BLink-able | Yes | Yes | Yes | No | No |
| Saves Preferences | Yes | No | No | Yes | No |
| Optimization | Exclnt | None | None | Poor | Fair |

Other features include: complete macro facilities, virtual file handling (for files larger than available memory), a choice of editor command sets (Wordstar or EMACS), and "a rich set of additional directives and pseudo-ops that can be very useful." (Dr. Gerald Hull, Amazing Computing, 6/88)

## You will believe an assembler can *FLY!*

INOVATRONICS, Inc.

The REAL POWER in Power Programming.
11311 Stemmons Freeway, Suite 8
Dallas, Texas 75229 214/241-9515

**only $89.95!**

*Note: AMA refers to the Amiga Macro Assembler, Copyright MetaComCo Plc. Inovatronics products distributed in the U.S.by Southern Technologies, American Software, and Computer Software Services. European Distribution: Burocare Ltd./U.K.Trademarks: Amiga: Commodores Int., Inovatronics and C.A.P.E.68k: Inovatronics, Inc., Devpac:HiSoft, Manx:Manx Software Systems, Inc., Assempro: Abacus, Inc., MetaComCo: MetaComCo Plc.

# Ray Tracing Fundamentals

## *The theory behind the magical images*

**by Cathryn E.F. Graham**

*Cathryn Graham was born in Calcutta, India and educated in Scotland. After working at Cambridge University with her husband Eric, they both relocated to the mountains of New Mexico to become full-time software developers. Cathryn specializes in software documentation and has written several software manuals including those for* ***Sculpt-3d*** *and* ***Animate-3D****, programs written by Eric. She is currently working on the design and testing of a new solid-modelling system for the Amiga.*

Ray tracing is one of the most effective ways of making realistic images of three dimensional scenes. If you have seen a computer generated image of solid objects showing shadows, highlights, reflections and transparency, then that image was probably made with the ray tracing technique.

The Amiga is the only modestly priced computer that lets you display a large number of colours on the screen at the same time. The HAM mode can display up to 4096 colours simultaneously, and that is enough to make quite realistic images. A number of ray tracing programs have been written for the Amiga, including the shareware program *DBW-Render*, *SSG* which was used to make the Juggler images, *Sculpt 3D*, *Silver* and *Animate 3D*.

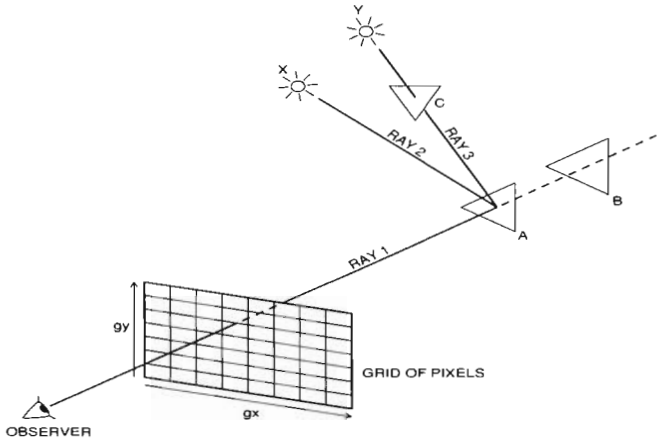



*Figure 1: Geometry for ray tracing*

### Representing objects

Before you can even think of making images, you must supply the computer with a representation of the three dimensional objects that make up the scene. Because a complicated shape, such as an elephant, is difficult to describe mathematically, it is much easier to build complicated objects from simpler ones. There are two different approaches: you can either build an object from solid volumes, or from surfaces.

The easiest volume to define mathematically is a sphere, since it can be defined in terms of the position of its center and its radius. Other shapes, such as rectangular blocks, cylinders, cones, etc. are almost as easy to work with, but unlike the sphere, you must also concern yourself with the orientation of the object. You can combine solid volumes in several different ways. The two objects can be added, or combined to produce a new object. Alternatively, one object can be 'subtracted' from another, for example if a cylinder is subtracted from a cube, the result is like drilling a hole in the cube. A third way of combining two volumes is called 'intersection'. The intersection of two volumes is a new volume that consists of the region that is inside both of the original volumes.

The technique of building objects from volumes is called Constructive Solid Geometry (CSG). It is excellent for modelling objects fabricated in a machine shop, because operations like sawing, milling, drilling and turning can all be easily represented. The Amiga programs *DBW-Render*, SSG and Silver all employ CSG.

One drawback with CSG is that while it is very good for making a model of a bracket or other regular shapes, it is not easy to make more irregular objects. Potential elephant modellers and others interested in building complicated objects usually describe their scenes in terms of a surface representation. The surface is built from a patchwork of triangular, quadrilateral or other polygonal faces.

The faces can be either curved or flat. If curved faces are used, then they can be blended together in a seamless manner to represent curved objects. Flat faces will result in the images having a faceted appearance. If the faces are made small enough, the approximation to a smooth curve becomes closer. As we will see later, there is a trick that can be used to make flat faces appear to be curved. *Sculpt 3D* and *Animate 3D* employ flat triangular faces for model building.

## Ray tracing

As its name suggests, ray tracing works by following light rays as they bounce around a scene. If the appropriate physical laws of reflection and illumination are used, the image of a simulated scene can have remarkable realism.

In addition to specifying the objects that inhabit our simulated environment, we must also supply an observer. We can regard the observer as a disembodied eye, as in Figure 1, or as a camera. In either case the observer has a number of properties:

1) A location relative to the other objects in the world.

2) A viewing direction. This can be represented as a pair of angles representing azimuth and altitude (pan and tilt, for photographers). Alternatively, the viewing direction can be specified by supplying a target position among the objects, requiring that the target position appear in the center of the image.

3) An angle of tilt. As photographers know, most pictures are best taken with the camera level with the horizon. If the observer is supposed to be aboard an aircraft, then it would be desirable to have the camera bank along with the aircraft, even though the horizon will not be level in the picture.

4) Zoom control. The angle of view must be specified by some means, for example by giving the focal length of the observer's viewing lens. This can be used to modify the effect of perspective. A wide angle of view will make perspective seem exaggerated.

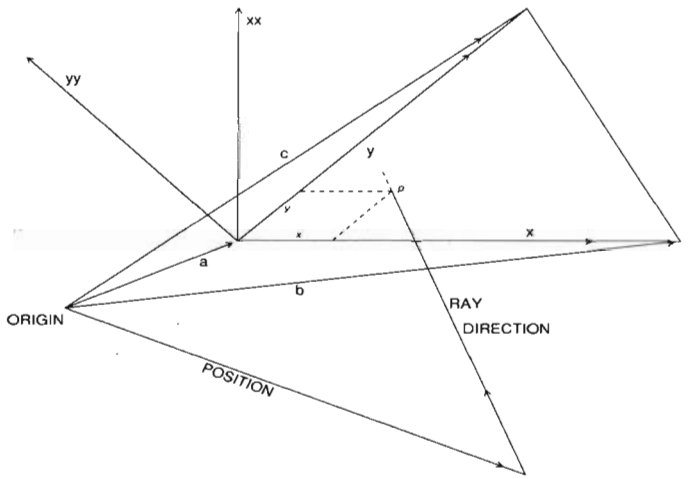



*Figure 2: Vectors used to calculate the intersection between a ray and a triangular object.*

Most of the light rays that bounce around a scene do not reach the observer, so they do not contribute to the image. The easiest way to select out the needed rays is to follow the light backwards, starting from the observer's position.

This is shown in Figure 1. A grid of pixels is imagined to lie in front of the observer. Each square on the grid corresponds to one pixel in the final image. The colour and brightness needed for a pixel in the final image is what the observer would see looking through the grid and out into the scene. A line, called a ray, is drawn from the observer in the direction that corresponds to a particular pixel. In Figure 1, RAY1 intersects with two objects A and B. Unless object A is transparent we can forget about object B, because it will be hidden from view. The brightness of the pixel will be the brightness of object A at the point where it was struck by the ray.

The brightness of a surface depends upon two things. The first is the reflective properties of the surface itself. If a surface is not shiny, then it reflects a fixed fraction of the light that strikes it with a particular colour. If we use a three colour system, with each pixel having separate brightnesses for red, green and blue light, the reflective properties of a surface can be described by three numbers representing the fraction of light reflected in each colour. This is just what we call the colour of things. For example, the values {0.95,0.90,0.05} would represent a yellowish object, while {0.21,0.18,0.23} would be a dark gray.

The second factor affecting the brightness of a surface is the amount and colour of the incoming light. This can be calculated by constructing rays from the surface to each of the light sources illuminating the scene. Each ray must be tested to see if it intersects with other objects. In Figure 1, RAY3 hits object C on the way to light source Y, while RAY2 to light X is not obstructed. If object C is opaque, then we do not have to include any illumination from light Y. The contribution from light X depends inversely as the square of the distance from light X to object A.

One other factor influences the way that a light source illuminates a surface, and that is the angle that the object presents to the light. Working in three dimensions, it is convenient to work with a direction that is at right angles to the surface; this direction is called the 'normal to the surface'. The illumination is diminished by a factor of the cosine of the angle between the incoming light and the normal to the surface.

## Bells and whistles

This simple description of the behaviour of light is capable of being used to make remarkably realistic images complete with shadows and shading. Our explanation of ray tracing has assumed that each object is opaque and reflects light uniformly in all directions. It is not difficult to add additional features.

If object A were semi-transparent, then object B would be partly visible. The standard ray tracing algorithm can be applied to the ray emerging from behind A to calculate the brightness contribution from object B. If object A represented the surface of something made of glass, then the ray could be deflected to represent the effect of refraction.

In similar fashion, if A were a mirrored surface, then a reflected ray could be calculated, and the ray tracing algorithm ap-

plied again. Real surfaces often have quite complicated reflective properties. Usually there is enhanced reflectivity when the angle between the ray to the observer and the reflection of the illuminating ray is small. This gives rise to the 'glint' that is seen with objects that are somewhat shiny, yet are not mirrors.

It is very easy to add additional optical effects to a ray tracing program. A few dozen lines of code can add a new physical effect, so it is often hard to avoid embellishing the code. This has lead to something of a cliche with regard to ray traced images. All too often, images of mirrored balls floating over a checkerboard plane are produced, whereas ray tracing is capable of generating almost any scene that could be imagined.

### Mathematics of ray tracing

A ray tracing program has to be able to make all kinds of geometric calculations in three dimensions. Recollections of high school geometry, and the difficulty of calculating the properties of even simple two-dimensional triangles, would make the mathematics of ray tracing seem like a daunting prospect. Fortunately, there is a better way of doing geometrical calculations in three dimensions. It is called vector geometry.

A vector is a mathematical abstraction that possesses both size and direction. A physical displacement can be represented by a vector, for example 'move 10 feet north'. To denote positions in three-dimensional space we will use a coordinate system. Each position can be represented by three numbers. All measurements are made from a single point, called the Origin. The numbers represent distances from the origin in three directions that are at right angles to each other, for example, eastward, northward and upward.

Once an origin is specified, any point in space can be represented by a vector representing the offset from the origin. Offsets can accumulate, so vectors can be added. The following C function performs vector addition.

```
vecadd (a, b, c)    /* vector addition a=b+c */
double a[3], b[3], c[3];
{
    int i;
    for (i = 0; i < 3; ++i)
        a[i] = b[i] + c[i];
}
```

Vectors can also be subtracted and copied:

```
vecsub (a, b, c)    /* vector subtraction a=b-c */
double a[3], b[3], c[3];
{
    int i;
    for (i = 0; i < 3; ++i)
        a[i] = b[i] - c[i];
}
```

```
veccopy (a, b)    /* vector copy a=b */
double a[3], b[3];
{
    int i;
    for (i = 0; i < 3; ++i)
        a[i] = b[i];
}
```

In the language of vectors, a regular number is often referred to as a scalar. A scalar can be multiplied by a vector to yield another vector:

```
vecsprod (a, s, c)    /* vector scalar product */
double a[3], s, c[3];  /* a=s*c */
{
    int i;
    for (i = 0; i < 3; ++i)
        a[i] = s * c[i];
}
```

Two vectors can be multiplied together to yield a scalar; this is sometimes called a 'dot' product. The code is quite simple:

```
double dot(a, b) /* dot product of two vectors */
double a[3], b[3];
{
    return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}
```

Note that a dot product of a vector with itself is just its size squared. A vector whose length is one is called a unit vector - it can be regarded as a pure direction. A regular vector can be converted into a unit vector as follows:

```
vecsprod(x, 1.0/sqrt(dot(x,x)), x);
```

The dot product has a geometrical interpretation: it is the product of the lengths of the vectors and the cosine of the angle between them. In particular, if two vectors are at right angles, then the dot product is zero.

There is a third way that vectors are involved in multiplication. It is called a vector product, because the result of the multiplication is also a vector.

```
vecprod (a, b, c)  /* vector product a=b^c */
double a[3], b[3], c[3];
{
    double t0, t1;
    t0 = b[1] * c[2] - b[2] * c[1];
    t1 = b[2] * c[0] - b[0] * c[2];
    a[2] = b[0] * c[1] - b[1] * c[0];
    a[1] = t1;
    a[0] = t0;
}
```

The vector product has the interesting property that the resultant vector is at right angles to both of the other vectors. Its

magnitude is equal to the product of the sizes of the vectors and the sine of the angle between them.

We now have enough of a vector toolkit to be able to describe almost any geometrical problem; in particular, we can set up rays. Remember that a ray is just a line that starts off at a particular point and heads off in a particular direction. It can be represented by the following structure in C:

```
struct ray {
   double position[3], direction[3];
};
```

A particular point along a ray can be calculated by the following function:

```
raypos (point, t, ray)
double point[3], t; struct ray *ray;
{
   vecsprod(point, t, ray->direction);
   vecadd(point, point, ray->position);
}
```

Defining a ray this way is sometimes called a parametric form, because the variable 't' is called a parameter. Different values of 't' will yield different positions along the ray.

Now let us set up a ray like RAY1 in Figure 1 that corresponds to a particular pixel position (i,j), for example. The following code fragments would do the job:

```
double viewdir[3];
/* unit vector for observer's viewing direction */
double obspos[3];    /* observer position */
double gx[3],gy[3];  /*vectors representing the pixel grid */
double d;            /* distance of observer from the grid */
int i,j;             /*pixel under consideration */
int nx,ny;           /* number of pixels in each direction */
struct ray ray;      /* ray to be calculated */
double temp[3];      /* temporary vector */

... /*  set up the above variables */
vecsprod(ray.direction, d, viewdir);
vecsprod(temp, i/((double)nx-1)-0.5, gx);
vecadd(ray.direction, ray.direction, temp);
vecsprod(temp, 0.5-j/((double)ny-1), gy);
vecadd(ray.direction, ray.direction, temp);
veccopy(ray.position, obspos);
```

The ray that we have calculated starts at the observer's location and passes through the specified point on the pixel grid. The next thing that must be done is to determine if the ray intersects with a particular object. If objects are triangles, they can be represented by the positions of the three corners:

```
struct triangle {
    double a[3], b[3], c[3], color[3];
} tri;
```

Let us divide the problem into two parts. First we will find the point p where the ray hits the plane that contains the triangle. Then we will see if p lies inside the triangle. If we calculate two vectors, X and Y, parallel to the sides ba and ca of the triangle, then we can use the vector product to calculate the unit surface normal, n, of the triangle (see Figure 2).

```
double p[3], n[3], x, X[3], y, Y[3], t, pminusa[3];

vecsub(X, tri.b, tri.a);
vecsub(Y, tri.c, tri.a);
vecprod(n, X, Y);
/*make it a unit vector */
vecsprod(n, 1.0/sqrt(dot(n,n)), n);
```

If the point p is in the plane of the triangle, then the vector (p-a) is at right angles to n, so:

$$(p-a)\bullet n = 0$$

where the "•" denotes a dot product . If the point p also lies on the ray, then:

$$p = position + t * direction$$

where the asterisk is used to denote multiplication of the scalar parameter t by the vector 'direction' of the ray. We can combine the last two equations and calculate the value of t that corresponds to a point on the ray that also lies in the plane of the triangle.

```
vecsub(pminusa, ray.position, tri.a);
t = -dot(pminusa, n) / dot(ray.direction, n);
raypos(p, t, &ray);
```

The point p now lies in the plane of the triangle, but does it lie in the triangle itself? To find out, let us express p in terms of the two vectors X and Y which are parallel to two of the sides of the triangle.

$$p = a + x * X + y * Y$$

Then we are describing p in terms of coordinates (x,y) in an oblique coordinate system whose axes are X and Y, but X and Y are probably not at right angles. Let us calculate two more vectors XX and YY that are both in the plane of the triangle, but XX is at right angles to X and YY is at right angles to Y. They can be calculated as follows

```
vecprod(XX, n, X);
vecprod(YY, n, Y);
```

By calculating the dot products of the expression for p with XX and YY, we can calculate the values of x and y.

```
vecsub(pminusa, p, tri.a);
x = dot(pminusa, YY) / dot(X, YY);
y = dot(pminusa, XX) / dot(Y, XX);
```

Finally, the point p is inside the triangle only if x and y are positive and x+y is less than half, so we could return the result of an intersection test like this:

```
return (x >0.0) && (y >0.0) && (x+y <0.5);
```

In a real ray tracing program, each ray would have to be tested against each object - the closest object would be the one with the smallest positive value of the parameter 't'. Next we would need to calculate the brightness of the object at point p. A lamp could be defined by the structure:

```
struct lamp {
   double position[3], brightness[3];
} lamp;
```

The vector from the object to the lamp and the distance from the lamp to the object can be calculated by the following code fragment:

```
double lvec[3], distance;
vecsub(lvec, lamp.position, p);
distance = sqrt(dot(lvec, lvec));
```

Although we have calculated the unit normal of the triangular object, we do not know which side of the triangle the normal is emerging from. If the triangle is part of a solid volume, it would be a good idea to arrange for the normal to point away from the object. Alternatively, we can ensure that the normal is directed towards the observer by reversing it if necessary, as follows:

```
if (dot(ray.direction, n) > 0.0)
   vecsprod(n, -1.0, n);
```

If the vector to the lamp is pointing in the opposite direction to the normal, then the lamp is illuminating the side of the object facing away from the observer, and the lamp does not contribute any brightness to the image at this point, otherwise we can calculate the brightness contribution to a pixel as shown:

```
double pixbrite[3], cosine;
int k;
cosine = dot(lvec,n) / distance;
for (k = 0; k < 3; ++k) /* loop through red, green & blue */
   if (cosine > 0.0)
      pixbrite[k] = lamp.brightness[k]
            * tri.color[k]*cosine
            / (distance*distance);
   else
      pixbrite[k] = 0.0;
```

The contributions from each lamp must be calculated, and allowance must be made for possible shadowing effects, by applying the intersection test along lines from the object to each lamp. Finally, some form of scaling, akin to an automatic exposure control, must be used to ensure that the pixel values are within the range accepted by the screen.

I hope this section has demonstrated that the mathematics of ray tracing is straightforward. By using a vector approach, three-dimensional geometry becomes quite tractable.

**Smooth shading**

I mentioned earlier that there were simple techniques for eliminating the faceted appearance of objects that are built from many polygons. One such method is called Phong shading, after one of the pioneers of computer graphics.

We have just seen how the brightness of a surface depends upon several factors:

1) The colour and brightness of the incoming illumination.
2) The distance from each light source.
3) The colour of the surface.
4) The angle between incoming light and the surface normal.

If two adjacent faces are the same colour, and both are far from the light sources, then the fourth factor contributes most to difference in brightness between the two faces. The surface normal changes abruptly as the observer's view passes from one face to another, unless they happen to be parallel.

If, instead of using the actual normal vector, you use an artificial one, you can achieve the illusion of smooth curvature. Figure 3 shows how this is done. At each corner, an average normal is calculated from the values for surrounding faces. Within each face, the fictitious normal is calculated by interpolating between the values at the corners of the face.

Anyone who has used the Phong shading facility in *Sculpt 3D* will be impressed by the way that coarsely constructed objects will seem to be smoothy curved. The illusion is only lost along the profile edge, when individual faces can be perceived.

**The economics of ray tracing**

There has to be a catch! Excluding the user interface, and the hooks into the operating system to display the image, the entire code for a ray tracing program including the effects of shading, shadows, reflection and refraction only seems to amount to a few hundred lines of C. The catch is the length of time required to render an image.

An Amiga HAM interlace overscan image has over 150,000 pixels. Typical *Sculpt 3D* scenes often contain more than a thousand triangular objects. With a few light sources, we have committed ourselves to almost a billion ray-object intersection tests. Each test requires many floating point arithmetic operations. A back-of-the-envelope calculation reveals that a single image may take a year or more to produce. Fortunately, there are many ways to optimize the ray tracing process.

Most intersection tests fail. If a scene has many objects, most objects will be small and the probability of an intersection will be low. If the program understands spatial relationships, then it

can dismiss many potential intersections as impossible. For example, if nearby objects are grouped in clusters, then a ray can be tested against the cluster as a whole, and if it misses the cluster, there is no need to test against each member of the cluster. The clusters can be organized in a hierarchy; many rays will fail the test with the topmost member of the hierarchy, so no objects need to be tested at all.

If objects are opaque, all that you need to find is which object the ray intersects first. By sorting the objects according to distance, substantial speed improvements are possible. Similarly, when performing shadow tests, it is enough to know that one object casts a shadow; you do not need to look for a second.

Another powerful technique for optimizing ray tracing is based on the concept of coherence. If one object contributes to the image at one pixel, it is likely to play the same role for an adjacent pixel. A similar technique can be applied to shadows: it is always worth testing the object that cast a shadow most recently, the odds are good that it will cast a shadow that will affect the next pixel.

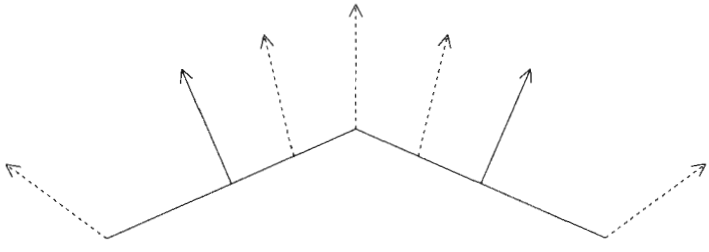
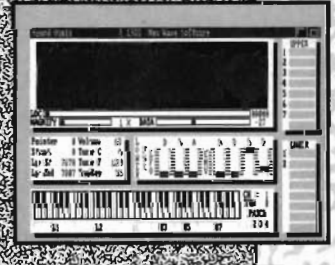


*Figure 3: Phong shading. The solid arrows represent true normals. Dashed arrows depict fictitious normals to produce the illusion of smoothness.*

**For more information**

Although ray traced images can be found in books and articles on computer imagery, the theory behind ray tracing is only given a brief mention in most textbooks. This may be because the theory makes a rapid transition from the elementary to the esoteric. If you want to learn more about the latest developments, you will have to consult journals such as those of the IEEE and the ACM.

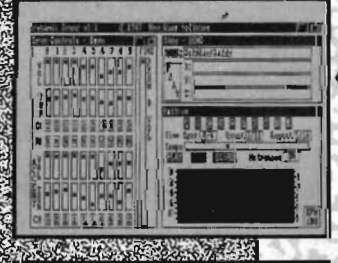
As a means of distributing images, software and articles on ray tracing and related topics, I am publishing a "Ray Tracing Newsletter". Each issue is distributed on an Amiga disk. Currently there are four issues available, and two more are in preparation. They are priced at $6.00 each, including shipping, or contact me directly at P.O. Box 579, Sandia Park, New Mexico 87047, for more information.

Sound Oasis

$99.95

The internal sound capabilities of the Amiga are better than that of any other personal computer. These capabilities mean nothing though, without quality digital sounds, which up till now have been scarce. Sound Oasis gives Amiga owners access to a large library of studio-tested digital samples, by using the Amiga's built in disk drive to read disks made for the Mirage Digital Sampling Keyboard. Sounds can then be played from a MIDI keyboard, the computer keyboard, or saved as an IFF standard file. Mirage is a trademark of Ensoniq Inc.



dynamic drums

$79.95

Transform your Amiga into a professional-quality drum machine with this software package. Easier to use than hardware-based drum machines because everything is displayed graphically on screen. Enter drum patterns quickly and easily in real time with visual feedback and editing. Create realistic drum tracks with any of the 100 drum and percussion samples that are included or use your own unique IFF one-shot samples. Dynamic Drums also has full MIDI implementation and even becomes velocity sensitive when triggered from a MIDI keyboard.

DYNAMIC STUDIO

$199.95

A powerful MIDI sequencer that takes full advantage of the Amiga's sound, graphics, and sophisticated user-interface. Dynamic Studio is perfect for professional applications due to its sophisticated editing capabilities and SMPTE support. It is also ideal for home studios, because in addition to sequencing MIDI instruments, Dynamic Studio has a built-in drum machine, and the ability to playback instruments translated with Sound Oasis.

New Wave SOFTWARE

No Copy Protection

P.O. Box 438 St. Clair Shores, MI 48080 (313) 771-4465

# Turbo Tracing

## *The real value in 68020 boards*

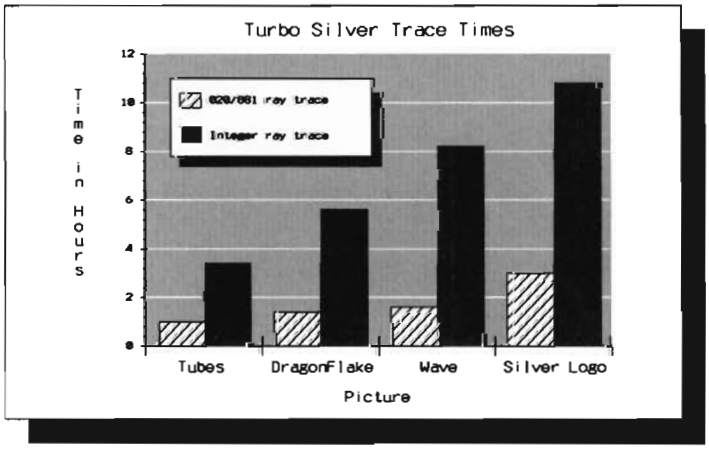
**by Louis Markoya**

*Louis Markoya is the creator of many well-known pieces of Amiga art, including "The Watching Angel", on the cover of TransAmi Volume 1 Issue 2. Louis also wrote the review of Photon Paint that appeared in that issue. He can be contacted on Compuserve at 75766,504.*

Most Amiga owners' pulses go up just slightly with the mention of 68020 boards and math co-processors. The dream of a faster, more powerful computer is in our blood. I know of at least three manufacturers of such boards, varying in price and features, but all relatively expensive. With this in mind, the practical aspects of owning such a board should be addressed. Just how fast will my computer be after installing this thousand dollar investment? (prices vary for manufacturers and speed of processors) What do Sieves, Dhrystones and Savage benchmarks mean to the end user?

Sure it all means your program may run faster, but exactly how much depends on many factors. Most 020's available for the Amiga run at a clock speed twice the speed of the standard 68000. This doesn't mean that everything will be twice as fast; graphics rendering done by the custom chips, for example, runs at the same rate. Some screen functions can be faster, however, (especially drawing functions that requires math) by virtue of the math being done faster.

There is little question that ray tracing is as computationally intensive a task as can be found for the Amiga. Using 4096 colours and the computing horsepower of the 68000, the Amiga is the first stock home computer that makes realistic traced images possible. On average, some four million calculations are made for a simple low resolution picture and easily 40 million when the objects get complex, or when glass and mirrors are used. And these numbers depend on the optimization techniques used by the ray tracing software, for these calculations would be 1000 times more without such optimization, and a ray-traced picture on the Amiga could take as much as a year to render. Luckily, the software manufacturers of Amiga ray tracing software have overcome the time problem by employing different optimization techniques.




Using ray tracing in much of my artwork, I have often wished for things to go faster, and I am always looking for the most powerful software. In addition, I have asked myself just how much it would help to get a 68020/68881 board. These questions would quickly be followed by more questions concerned with software and hardware compatibility. This article addresses these questions, most importantly that of speed... how much faster do things get?

As a forum I used Turbo Silver, from Impulse. Though the finished product is not on the shelves, the interim versions are available as upgrades to any registered user, and Impulse has made a version specifically optimized for the 020/881 combination. As ray tracers go, Turbo Silver is one of the most comprehensive packages available, offering *many* advanced features found only in professional packages (a full review will follow in an upcoming issue). Some of these features will be discussed briefly so you can better realize the tasks set for the computer in rendering the images I have created for this test.

As for the 020/881 board, I've been testing the new Hurricane 2000 model, by Ronin Research and Development Inc./ Finally Technologies. The 68020 is clocked at 14.32 MHz and the 68881 is clocked at 16 MHz. By using a separate crystal, any speed 68881 could be used, and with faster math co-processors even greater speed benefits could be realized. Although the board was one of the first shipped, its design was complete and solid; no

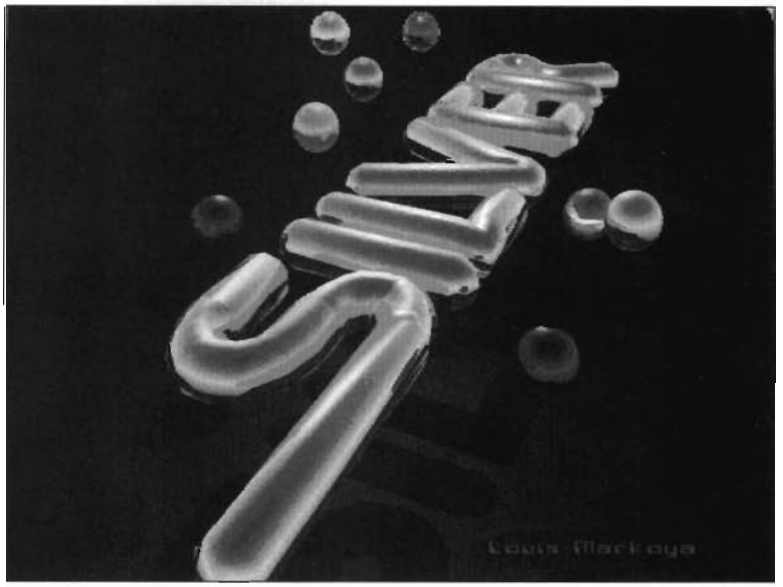
jumpers and no complications. It installed easily and worked flawlessly with both the 2090 Hard Disk controller and the 2052 Ram expansion units I own. According to Ronin, the Hurricane board uses proprietary fast bus synchronization to offer a performance increase over competitors. In simpler terms, they have found ways to address the Amiga 16-bit bus in less cycles than competitor's boards.

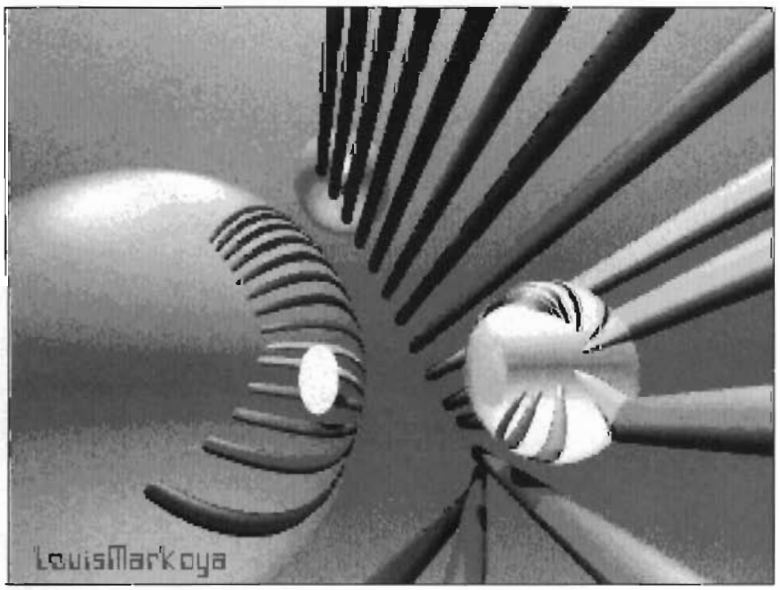
The premise for the experiment was to set up several scenes of varying complexity to trace. Knowing integer Turbo Silver already had substantial speed advantages over its competitors, I was eager to see what differences floating point would bring. (Turbo Silver's speed can be attributed to the coding experts at Impulse and using Octree volume methods, which test space for and collision of a light ray with an object before the calculations are done.) It is also important to note that the actual quality of traces with the Floating Point version is slightly better, due to the higher math precision. This will be of greater benefit (and easier to visualize) with the introduction of 24-bit frame buffers, which Silver's file format already supports. In both the floating point and integer versions, all scene parameters were kept the same. Light positions, objects, reflectivity, etc., were identical. Several techniques were discovered that are contributing factors to trace times and are applicable to all ray tracers. In particular, the placement of lighting can make vast differences in trace times (up to 300 percent!), and in many cases with little change in the resulting image. Following are descriptions of the scenes and images, plus the results of the Standard Amiga vs. Hurricane 020 Trace times (In Hours : Minutes : Seconds).

Note that "The Dragonflake" and the Silver logo are interlaced HAM pictures, while the others are non-interlaced. This plays a major role in trace times.

The Silver logo consists of tubular letter objects consisting of over 2000 polygons. They were given the quality of chrome and the recursion level set to 13. (This tells Silver to trace the beam's reflection 13 times back and forth before quitting). Several highly reflective spheres were added to approximate the Silver Logo.

**Integer time: 10:48:20**
**020/881 time: 3:02:15**
**3.5 x increase**




The Tubes trace was meant to be a relatively simple image, using reflection and refraction. The image consists of several multi-colored tubes and three spheres, two of which are mirrored, and the third glass. In this trace the Anti Aliasing was set high to avoid drastic staircasing on the upper tubes; this setting increases computation time.

**Integer time 3:35:27**
**020/881 time 0:58:04**
**3.7 x increase**

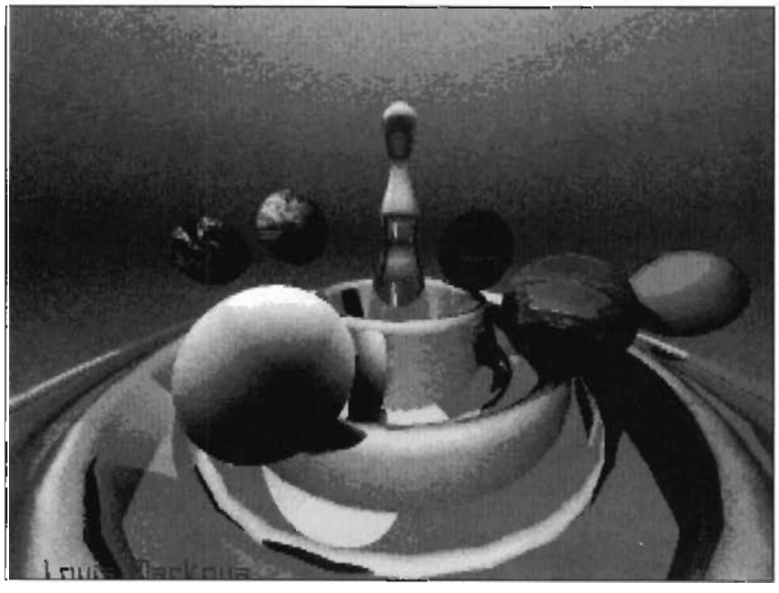
The graph on the first page of this article depicts the amazing performance increase with the Hurricane board. With this level of speedup, one can drastically cut production times and perfect the traces you dream up. Further speed increases of about 100% are expected with the addition of 32-bit RAM (to be tested and reported on later). For the professional and the enthusiast, this may more than offset the pricing question.

As for compatibility, the Hurricane board worked flawlessly. Virtually none of the software I tested (Word processors, Music, Painting, Comm, etc.) had any problems (though I'll admit I never booted a single game during the review). And there wasn't the slightest hint of hardware incompatibility. I have to admit, it has been great.

In this article I have tried to demonstrate the real value of owning a speedup board. I can highly recomend both versions of Turbo Silver and the Hurricane/Ronin Reasearch board. For those of you who want to compute in the fast lane, this is the way to go.

Turbo Silver
Impulse
6870 Shingle Creek Pkwy #112
Minneapolis, MN 55430
(800) 328-0184

Hurricane 020 boards
Ronin Research and Development, Inc.
P.O. Box 1093
Alameda, CA 94501
(415) 769-9325

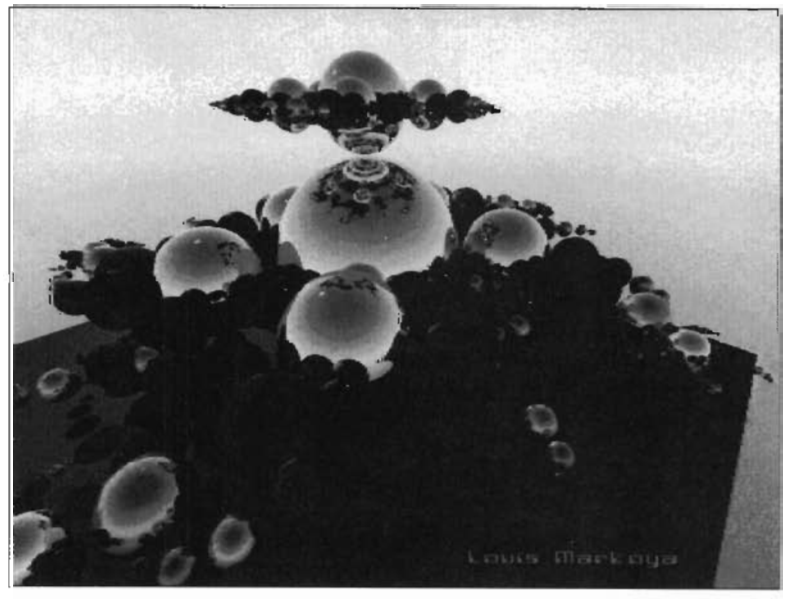


The Wave features a complex object with over 1500 polygons that makes up the "frozen in time" drip in this picture. In addition, it has five of its six spheres mapped with IFF files of fractals. The images are wrapped in various manners, a computationally intensive operation. The Drip was given highly reflective attributes and a recursion level of 13.

**Integer time 8:08:30**
**020/881time 1:42:05**
**4.8 x increase**

The Dragonflake is an object I made to resemble a Dragon-like fractal, and a snowflake. It consists of over 800 highly reflective spheres. These were given the index of refractor of Silicon and a recursion level of 8. (Turbo Silver allows the user to define a specific index of refraction, as well as the filter values, in Red, Green and Blue.)

**Integer time 5:45:02**
**020/881time 1:27:33**
**3.9 x increase**



# INTERCHANGE ™

**Share objects between Sculpt3D, VideoScape 3D & Forms in Flight**

**NOW YOU CAN...**

- **Use Sculpt 3D to ray-trace VideoScape 3D objects**
- **Do Forms in Flight animations on Sculpt 3D objects**
- **Create VideoScape 3D objects using the Sculpt 3D interface**

**Full Intuition interface for all InterChange functions**

Object Disk #1 now available. Includes a Sculpt 3D font, plus lots more Sculpt 3D and VideoScape 3D objects!

InterChange master program plus Sculpt 3D and VideoScape 3D Conversion Modules, $49.95. Forms in Flight add-on Conversion Module, $19.95. InterChange Object Disk # 1, $19.95.

This product requires objects from Sculpt 3D and/or VideoScape 3D and/or Forms in Flight. It is not a standalone animation program.

To order, send check or money order. Please include $3.00 postage & handling. MA and WI residents add 5% sales tax. InterChange is a trademark of Syndesis. Sculpt 3D, VideoScape 3D and Forms in Flight are trademarks of Byte by Byte Corporation, Aegis Development and Micro Magic respectively.

**SYNDESIS**
20 WEST STREET
WILMINGTON, MASSACHUSETTS 01887
617-657-5585



# Fish Disks with TransAmi Labels

...are an exceptionally valuable resource. Each program on a disk is displayed prominently on the label, usually followed by a brief description. Below is a sample of one of the labels.

**Fish Disk #103**
From Fred Fish's Library of Freely Redistributable Amiga Software

| | |
|---|---|
| **AvlTrees** | Library & test program for AVL trees w/source |
| **Calc** | Shareware RPN calculator - V1.0 |
| **Cref** | C cross referencer w/source |
| **DosKwik** | Shareware fast loader |
| **IntuiDOS** | Manipulate files w/mouse |
| **MFF-Update** | Text import for MicroFiche Filer |
| **Pack-It** | Shareware: Pack all files/dirs on disk into 1 file |
| **Sol** | Solitaire game w/source |

All Fish Disks you order from us come with labels, and if you already have the disks, you can order the entire set of labels (currently up to disk #128) for just $25.00 US.



**The fastest**
# Modula-2
**Software Development System for**
**AMIGA**

**$ 249**
**Demonstration disk $ 10**

Modula-2 is the successor of Pascal. Both languages were invented by Professor Niklaus Wirth. Modula-2 is an all purpose language suited for educational, home and professional use. Modula-2 leads to good programming technique, easy readability and maintainable programs. In fact, a lot of schools and universities have chosen Modula-2 for their programming courses.

M2Amiga is a software development system for Modula-2 based on an extremly fast single pass compiler. It is fully integrated into Amiga Workbench and is easy to use from CLI or Workbench. It runs on all Amiga models with a minimum configuration of 512k RAM and one disk drive.

M2Amiga has been developed specially for the Amiga and therefore optimally supports the features of this unique computer.

- Single pass compiler produces optimized native MC68000 code from standard ASCII files. Full support of latest edition of Modula-2.
- Fastest library calls possible on the Amiga. No intermediate code needed for library calls and parameter passing.
- Supports FFP, 32 bit IEEE and 64 bit IEEE reals formats, you can even mix all three formats within the same program.
- Easy access to MC68000 registers, allows the insertion of inline code.
- Full interface definition to the Amiga's operating system following the official definitions.
- The best runtime system currently available on the Amiga. Opens and closes all used libraries, captures all GURUs and releases all used resources after program crash or termination. Routines for System requesters enable user interface consistent with the Amiga's Intuition.
- The emacs based editor places the cursor on compilation errors and explains them in full text.
- In just a few seconds the linker produces stand-alone applications that can be run from CLI or Workbench. Stand-alone applications do not contain any useless overhead – they are compact and fast.
- All programs can produce icons for their output files to allow effective use of the intuitive Workbench environment. Experience the new feeling of program development by selecting and double clicking icons!
- Also included are a project management tool and other utilities.
- Besides the handy manual, an introductory guide to Modula-2 is included. This documentation allows both the beginner and the advanced user to find any information needed quickly and easily.
- Demo programs show the use of various features of the Amiga.
- For professional program development special utilities are available.
  + Source Level Debugger, the new way of debugging your programs. Inspect all your variables (displayed in their own format), zoom into structures as records or arrays or follow pointer chains by clicking the mouse buttons
  + Object file converter, converts standard Amiga object files into M2Amiga format. This allows linking of routines written in other languages.
  + Linker for devices or libraries. Add power to the Amiga by supplying new libraries and devices. Easy and intuitive programming instead of fooling around with Assembler and Amiga linker are possible now!
  + Modula-2 Amiga Programming System Environment

We have Modula-2 Compilers for HP-UX, IBM/370, PCs (Taylor, M2SDS, JPI), OS-9 and Sun, tomorrow even more!

Texas residents add 8% sales tax. International Orders add $ 20 shipping.

**The Modula-2 people:** INTERFACE TECHNOLOGIES

**3336 Richmond, Suite 323, Houston, TX 77098-9990 (713) 523 8422**
**Dealer inquiries welcome**

2U

**A. + L. Meier-Vogt**
**Im Späten 23**
**CH-8906 Bonstetten/ZH**
**Switzerland**
**Tel. (41) (1) 700 30 37**

# "LUCAS" For the Amiga 1000

## *A 68020/68881 platform board*

**by Brad Fowles**

*Brad Fowles is a design engineer at Anakin Research, and is responsible for the hardware design of Anakin's Easyl drawing tablet. He has been involved with the Amiga since the machine's debut in 1985. Brad tells us that one of his guiding design principles for computer hardware is, 'Don't let out the magic smoke'. Be careful of this if you attempt this project.*

Most of you in the Amiga community are well aware of the wonderful software available in the Public Domain. As a hardware type I have often been envious of the ease with which software can be shared among developers and users alike. Ideas and techniques can be distributed through BBS networks to the general benefit of all. In contrast, hardware developers lead a comparatively solitary existence, the exchange of ideas impeded by economic and logistical problems.

Can there be such a thing as Public Domain Hardware? Obviously no one can give away printed circuit boards, but perhaps we can do the next best thing: give away as much information as possible and make bare PCBs available for as close to cost as shipping allows.

The project is a platform board called LUCAS (Little Ugly Cheap Accelerator System) which replaces the 68000 in your Amiga 1000. LUCAS provides greater system performance and allows the use of the 68881 math coprocessor as well as an upgrade path to 32-bit wide memory.

The board has a 68020 and 68881 running at 16 MHz, and interface logic (consisting of four PALs, four discretes, a 16 MHz crystal, two SIP resistor paks, and some capacitors) to transpose 68020 cycles to 68000-like cycles. LUCAS also has a connector which will allow you, at a future date, to add 32 bit wide memory. (I'll try to get the fine people at *Transactor* to publish a memory board for this system in a few months.)

You can order the bare printed circuit board for $40, and the four-PAL chip set for $25 (see ordering information at the end of this article). The rest is readily available from local suppliers. The schematic and PAL equations are published here. Anyone who wants the film plots or Net lists so they can adapt the form factor to the Amiga 500 is welcome to them for whatever it costs me to get and ship them to you. (PCB design was done using P-Cad on er... an AT ( ...almost said the I word )).

If you own an Amiga 1000 and you would like to experiment with a 68020 and 68881 combination to improve performance, this may be the cheapest way to get there. Unfortunately, the chip set is going to cost you about $370.00 Canadian. Our aim is to make the rest as cheap as possible. You should be able to be up and running for under $475.00, or about three quarters of that if you live in the real world.

I decided to do this project for three reasons. One, I wanted one myself and couldn't afford the commercial versions. Two, some friends of mine who are using *Sculpt 3D* and *Animate* from Byte by Byte (both are available in 68020-68881 versions) needed more horsepower to render their images fast enough to actually make money at it. Three, I figured all of us Amiga 1000 owners out there with true hacker's hearts needed some light in our future since 1 meg of chip ram ain't.

When I started the design of this board, I used as a reference an article from EDN, January 9th 1986, pp 216-219. While looking at this design I became aware of an application note from Motorola AN944/D, *MC68020 and MC68881 Platform Board for Evaluation in a 16-bit System.* I recommend both these documents, especially the latter, if you wish a better understanding of how this board works. Unfortunately it is impossible within a short article such as this one to give more than a brief overview of how the board works. In the technical section of this article, I will try to highlight those aspects that are specific to the Amiga, but a thorough understanding will require some digging on your part. I also recommend the User's Manuals for the MC68020 and the MC68881 which are available from Motorola as "MC68020UM/AD" and "MC68881UM/AD" respectively.

Okay, Here is the disclaimer: If you get one of these bare boards and carefully put it together and then install it into your Amiga, you should have no problem and you'll be up and running in an evening or two. If you have problems, then it's up to your ingenuity to solve them. If you don't have some experience with a soldering iron, please, don't let this be your debut. I will gladly help anyone with problems. There are three ways you can get in touch with me: USENET at anakin@gpu.utcs.toronto.edu (Brad Fowles), BIX in the ANAKIN, AMIGA conference or by regular mail through *Transactor.* If you do manage to get my phone number you'd

better be able to sweet talk me within 30 seconds. I hope that if there is sufficient interest out there that local user groups or individuals will add their help to anyone having problems. I have no objections should anyone get the bare boards and put them together and install them for a modest price, but please remember that the purpose of this is to make these available to end users as cheaply as possible.

If I haven't scared you off, please read on. If I have, well... so long, and thanks for all the fish.

Once you get one of the bare boards and procure all the parts, follow the enclosed instructions and carefully solder sockets for all the ICs and the crystal onto the board. Solder the resistor paks and the capacitors into place. Insert the 64 Pin header for the 68000 socket and solder it in.

Installation is quite simple but should be carefully done. Remove the plastic cover and the EMI shield from the Amiga base unit. On the right hand side of the PCB, just beside the Expansion connector, is the 68000 CPU. Gently pry the 68000 out of its socket, and store it on a piece of styrofoam somewhere safe. Now insert the LUCAS board into the 68000 socket, being careful to ensure that all 64 pins are correctly inserted into the socket. If you want to be really careful, remove the disk drive so you can see better. Watch the ribbon cable for the internal disk drive, as the bends in the cable can make things awkward. As long as you're careful and don't force anything, you should have no problem. You can do initial tests with the cover off, but once you're satisfied it's working put the base unit with its EMI shield back together again. That's all there is to it. Your heart can now resume normal operation.

You don't need to know a great deal about the inner workings of the LUCAS board to enjoy using it, but for those who would like a better understanding of the nature of running a 32-bit 68020 in a 16-bit 68000, read the section at the end of the article to get the key technical points.

**Benchmarks**

To give some idea of the performance improvement you can expect with the 68020-68881 pair, I have used four programs Al Aburto made available on the DEVCON disks distributed at the Washington Developers' Conference. These benchmarks were run on an Amiga 1000 with a 2 Meg Microbotics Starboard memory board and a Comspec 20 Meg Hard Disk. The operating system was Kickstart 1.21 and Workbench 1.3 Gamma 7. It should be noted that when the 68020-68881 pair is installed, the new IEEE math libraries which support the 68881 are used for floating-point transparently. I ran these benchmarks first with a standard 68000 and then with the LUCAS board.

Savage:

68000 470.0 sec. Error -6.9e-7
LUCAS 14.5 sec. Error -5.7e-4

Whetstone:

68000 24 kwhets/sec.
LUCAS 126 kwhets/sec.

Calcpi:

68000 4.85 kflops/sec. Error -1.39e-11
LUCAS 11.9 kflops/sec. Error -2.78e-11

Float:

68000 10000 iterations 45.74 sec.
256000 iterations 286.96 sec.
LUCAS 10000 iterations 12.80 sec.
256000 iterations 118.56 sec.

Of course, speed could be further enhanced by using inline F instructions for the floating point stuff, and even further enhanced by using 32-bit wide no-wait-state static memory.

Please remember that benchmarks are like political speeches; they only seem to make sense.

**Software Considerations**

Most software runs just fine on the 68020 but there are some programs which will guru on you. One of the major reasons for this is that on the 68020 all the instructions that are on the 68000 are implemented with the inevitable exception of one: the MOVE SR <ea> instruction. On the 68000 this is a user mode instruction; on the 68020 (and 68010 and later parts) it is a supervisor mode only instruction, i.e., if it's executed on a 68020 in an Amiga, you get a privilege violation guru.

If you're writing software, don't use this instruction; use instead the GetCC() library function which translates to a MOVE CC <ea> on the 68020, which is a valid user mode instruction. This function translates to a MOVE SR <ea> if there is a 68000 in the Amiga. This way you're safe both ways. (To run existing software that fails on the 020 for the above reason, you can solve the problem by first running a program called DeciGel that traps the offending instruction and does the right thing. DeciGel can be found on the *TransAmi* disk for this issue, or on Fish disk #18.

If you're one of those people who thought encoding information in the upper eight bits of the address field was a nifty idea ... Oh well, time to learn the error of your ways.

Of course, if you use any instructions from the 68020 superset then this code will never run on a standard Amiga. For further information, see section 21 of the Washington Amiga Developer Conference Notes, *Software Issues in 32-bit Amiga Systems* by Dave Haynie.

The new release of 1.3 has new IEEE Double Precision Math Libraries which take advantage of the 68020-68881 pair if it is present, and can immediately speed up any existing programs which use the math libraries.

If you want blindingly fast floating point, the best way is to recompile your code so that it uses direct inline F instructions. On the disk that comes with the PC board, you'll find programs called Mandslow and Mandfast. They are slight adaptations of RJ's original Mandelbrot program, adapted by Eric Haberfellner. Both programs are the same except that Mandslow was compiled for a standard Amiga, while Mandfast was compiled to use inline F instructions. Using Mandfast, a moderately deep Mandelbrot that runs in 1 hour 20 minutes on a standard Amiga runs in 4 minutes 20 seconds with the LUCAS board installed.

### Compatibility

The LUCAS board works with all the expansion boards I have, but I'm sure there will be some out there that will bomb out. I will keep a list of those that do and those that don't and post it regularly on Usenet and BIX. The ones I have are the Comspec 20 Meg hard disk, Comspec 2 Meg Memory board, EASYL, and the Microbotics Starboard 2 Meg Memory board.

As a matter of interest only, the board works fairly well at 20 MHz, but periodically bombs. I have only 16 MHz parts; when I debug the bomb it seems to be the fault of the on-chip instruction cache. If you have 20 MHz parts, try it and let me know. Even if you have 16 MHz parts, it's worth the price of a 20 MHz crystal to see if it will work. Who knows? You might get lucky.

### Conclusion

The performance of the Amiga 1000 with the LUCAS board installed will be improved, but it won't perform miracles. For general purpose computing, I find that compiles are only about 1.4 times faster, hardly worth the trouble. However, any program which uses floating point will be improved considerably, and those which have embedded F instructions will indeed appear miraculous.

On the other hand, the board does allow for 32-bit wide expansion memory, and if installed you can expect considerable general purpose performance improvements as well. I plan to design two boards: one with standard 100 or 120 ns DRAMs and a second with some high speed static RAM for no-wait-state operation at 16 MHz. You get most of the performance increase by having the memory 32 bits wide, but I can't resist seeing how fast it will go with no wait states at 16 MHz.

Stay tuned to *TransAmi* and the Nets for updates. Enjoy!

* * * * *

*Brad is making the bare board and the PAL chips for this project available at a nominal cost: $40 for the board (which comes with a disk), and $25 for the four PALs. You can order directly from Brad at the following address:*

*RR #5 Caledon East*
*Ontario, Canada*
*L0N 1E0*

# Technical Discussion

Once the LUCAS board has been installed, we essentially have divided the CPU time into two discrete blocks: One, seemingly operating at 7.16 MHz and synchronous to the special purpose chips responsible for the video, sound, etc. and two, a 16 megahertz asynchronous system between the 68020 and 68881 and any possible 32-bit wide memory connected to the LUCAS bus.

The essential design criteria I used for the board were that it should be able to run asynchronously to the Amiga clock (so speeds of 16 MHz or greater could be achieved) and that there be no connection other than through the 68000 socket (to simplify installation.)

In order to achieve this, the board must look like a 68000 (4 clock standard bus cycle) running at 7.16 MHz when it is running its bus cycles, but when it is doing internal processing or talking to the MC68881 or future 32 bit wide expansion ram, it should run at the full 16 MHz (3 clock bus cycle).

90% of the problem in making this board work comes down to the problem of making the 68020 appear exactly like the original 68000 it replaces as it has been used architecturally in the Amiga, but able to go like stink when it gets the chance.

The address and data lines are easily implemented, as they are connected directly from the 68020 to the 68000 socket. Note that the 16 data bits are connected to data bits D16 through D31. The upper eight address bits on the 68020 are simply left unconnected.

I have used the * convention to indicate low true signals for ease in typesetting the article, i.e., *AS means AS is a low true signal. The PAL equations are written in CUPL format so I apologize to all you PALASM users.

### 68020 to 68000 Interface

The 68000 has an asynchonous bus structure. It asserts Address Strobe (*AS) to begin a bus cycle then waits for the assertion of *DTACK to end the cycle. This is usually 4 or 6 cycles, but may be held off by some peripheral device. The 68020 works much the same way except there are two *DTACK-like signals, *DSACK0 and *DSACK1. Because the 68020 can address in bytes (8 bits), words (16 bits) and longwords (32 bits) it must be able to differentiate between them. It does this by use of its dynamic bus sizing capability. A peripheral responds to a bus cycle by asserting one or both of the *DSACKx signals which tells the 020 the size of the transfer.

| DSACK0 | DSACK1 | TRANSFER SIZE |
|---|---|---|
| 0 | 0 | 32 bit transfer |
| 1 | 0 | 16 bit transfer |
| 0 | 1 | 8 bit transfer |
| 1 | 1 | Insert Wait States |

Bus cycles on for the Amiga are always 16 bits wide so we will assert only *DSACK1 when responding to Amiga cycles. When we are running cycles for the 68881 (FPU) or 32-bit wide RAM on the LUCAS board expansion connector we must assert the appropriate *DSACKx combination.

In general terms with no wait states the 68000 will run a bus cycle in 4 clock cycles; the 020, however, will run the same bus cycle in 3 clock cycles. To correct this we must delay *AS and *DS (Data Strobe) from reaching the Amiga until after the rising edge of the S2 phase of the 7.16 MHz CPU clock. This is accomplished by the flip-flops U8a and U8b: inverting *AS from the 020 and using the complementary output with the flip-flop's reset tied to the inverted *AS will delay *AS the desired amount and terminate *AS20DLY when the *AS from the 020 terminates. This same technique is used for *DS. This creates the two timing signals *AS20DLY and *DS20DLY.

Byte addressability on the 68000 is accomplished by the Upper Data Strobe (*UDS) and the Lower Data Strobe (*LDS). The 020 has only a single Data Strobe (*DS). It uses a combination of the two SIZE pins and A0 and A1 to define the transfer pattern from the 020's internal multiplexer to the external data bus. (Note: bytes appear on data bits 24-31, words appear on data bits 16-31). It is therefore necessary for us to create *UDS and *LDS. This is accomplished by the following PAL equations. Note: The data strobes are not asserted during a Coprocessor cycle. (CPCS)

```
!UDS = (!DS20DLY) & (!A0)   & (CPCS)
!LDS = (!DS20DLY) & ( SIZ1) & (CPCS)
(!DS20DLY) & (!SIZ0) & (CPCS)
(!DS20DLY) & (  A0 ) & (CPCS)
```

The 68000 contains logic to support the 6800 family of products, and the Amiga uses this to interface to the 8250s. We must also emulate this interface as it is not present on the 020. A secondary clock called the E clock must be generated. It has a frequency of 1/10th the CPU clock and has a duty cycle of 60% low and 40% high. This is done by a decade counter in PAL U4. When running a 6800 family cycle the Amiga or peripheral generates a Valid Peripheral Address signal (*VPA). The 68000 then syncs itself with the E clock and issues a Valid Memory Address (*VMA) and ends the cycle on the falling edge of the E clock. The equation,

```
!Z3 = !QD #
QC #
QB #
QA ;
```

on PAL U4 in combination with the equation

```
!Z1.D = (DS20DLY) & (!Z1) #
(DS20DLY) & ( Z3) & (!VMA);
```

asserts *DSACK1 in the 9th state of E clock by the generation of the Z1 signal so that the long VPA, VMA cycle can be terminated correctly.

**68020 to 68881 Interface**

The MC68881 chip select (*CS) must be decoded from the 020. The 020 generates a 111 on the Function Code pins (CPU Space), a 0010 on the address lines A16-A19 which means this is a FPU coprocessor cycle, and a coprocessor ID on Address lines A13-A15. Since there is only one coprocessor in this design, A13-A15 are undecoded. The rest is decoded by PAL U4 in the following equation:

```
CPCS = (FC2) & (FC1) & (FC0) & (!A19) & (!A18)
 & (A17) & (!A16)
```

This generates the *CS (Chip Select) to the 68881.

**Zen and the Art of Cycle Termination**
**(An Asynchronous Synchronous Asynchronicity)**

The generation of the *DSACK1 signal from the Amiga *DTACK caused me at times to doubt not only my own sanity but that of the universe itself. The *DTACK signal from the Amiga should appear and be sampled during the S4 phase of the clock cycle. Unfortunately it doesn't quite know that. It responds more or less correctly when it is talking to internal RAM but when external (fast) RAM is accessed, *DTACK comes back almost right away. Remember that *DTACK is the only way we have of determining the length of a cycle. We will cope with this anomaly in a moment.

Since the 020 is operating at 16 MHz - i.e., quite asynchronous to the Amiga clock - you have to sync up somewhere along the line with the Amiga 7.16 MHz clock. The ideal place to do this is when the two Amiga clocks C1 and C3 are in the condition C1 high and C3 low. These signals are not available at the processor and for a long time I had these two lines coming up off the motherboard. However the 7.16 MHz clock that is available at the processor can produce a reasonable facsimile. I divide the 7.16 MHz clock by two using U9a then logically OR it with the original 7.16 MHz clock and this turns out to have the same timing as C1 high and C3 low (my faith in the universe began to rekindle at this point.)

In the PAL equations this is DTPRELIM (DTack PRELIMinary). Now we have a reference point to sync back up with the Amiga.

In a saner world the combination of *DTACK and the Z1 signal (for termination of VPA, VMA cycles) would be sufficient to create the term SYSDSPRE1 (SYStem DSack1 PREliminary 1), but we have to delay till *DTPRELIM is true to sync up with the Amiga, plus cope with the quick response of *DTACK anomaly when talking to fast RAM And sync back up with the 16 MHz 68020 when we do finally issue a *DSACK1.

**Confused? Wait! It gets better**

Most dynamic memory boards, when connected to the Amiga expansion bus, will assert XRDY to hold off the assertion of

*DTACK while they do a refresh cycle. This puts in enough wait states so that the memory board can complete a refresh cycle. The problem is, soon as XRDY is asserted, a 20-30 ns glitch occurs in *DTACK, prompting the 020 to terminate the cycle before the data is even thinking about arriving on the bus. The solution is to avoid decoding it till the S4 phase of the Amiga 7.16 MHz clock. I delay *AS20DLY again for *DTQUAL and again for *DTQUAL1. DTQUAL becomes part of the *DTACK term and *DTQUAL1 is wired to QUAL (I needed the 7 ns across the PAL) then QUAL is added to the *DTACK term, giving:

```
!SYSDSPRE1 = !Z1#
(!DTACK)&(!AS20DLY)&(!DTQUAL)&(!QUAL)
```

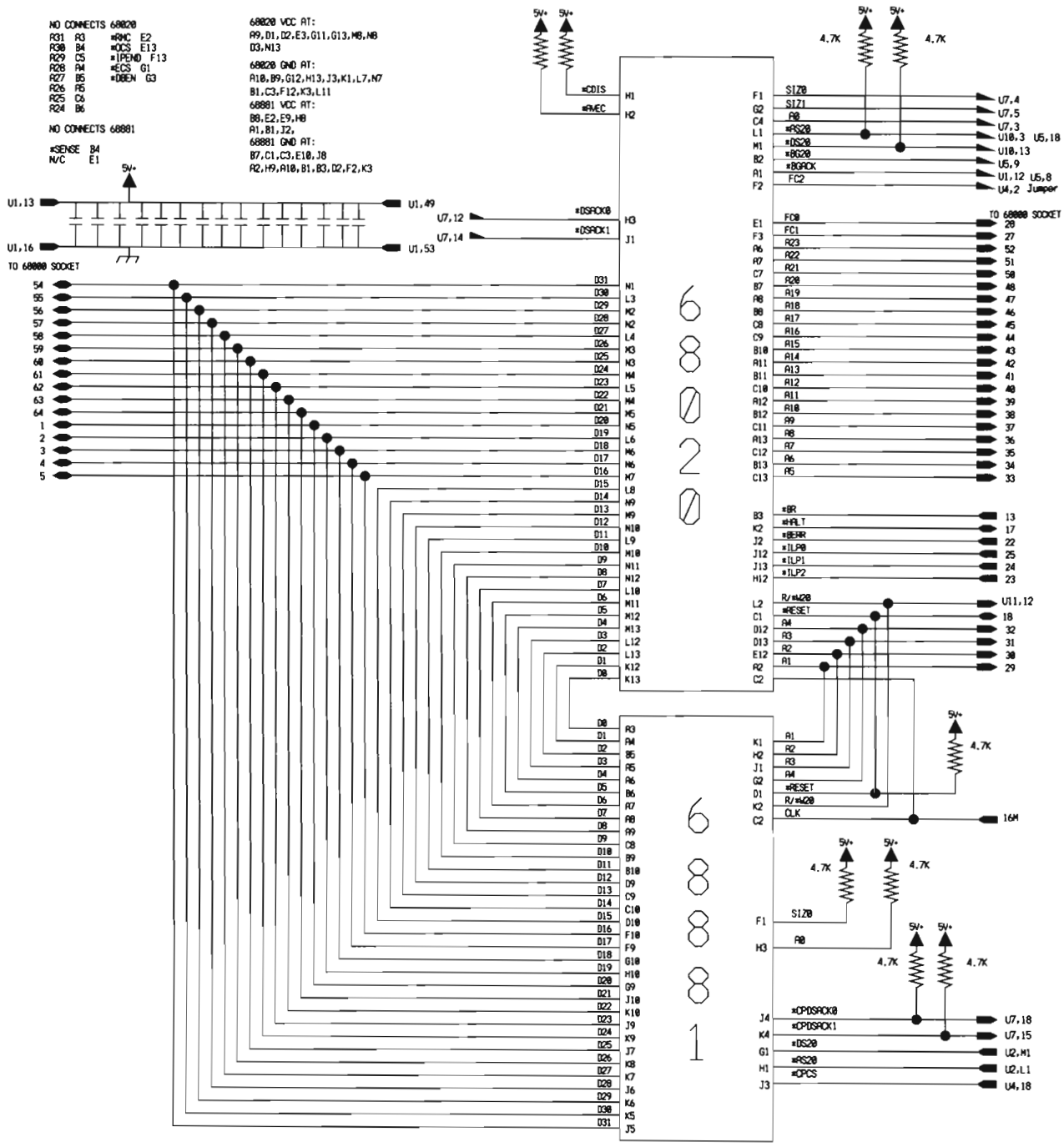
This solves the quick *DTACK problem. We buffer this (another 7 ns) by:

```
!BUFOUT = !SYSDSPRE1
```

Add in the Amiga syncronizing term

```
!DTTRIG = (!BUFOUT) & (!DTPRELIM)
```

Now we have an edge which is syncronous to the 7.16 MHz Amiga system. We then use this to trigger a Flip-Flop which has patiently been waiting for all this tomfoolery to end, and will ship !SYSDACK1 to yet another Flip-Flop to sync it back up to the 16 MHz 020 clock, and then to the awaiting PAL U7 for



additional decoding. I feed the Flip-Flop U9b with *ASDLY so that the !SYSDSACK1 signal will terminate when *AS does.

We're almost done.

PAL U7 then combines *SYSDSACK1 with the 68881 *DSACK1, CPDSACK1, and *SRDSACK1 (which comes from the expansion connector for future Static RAM), and finally and enthusiastically begets *DSACK1.

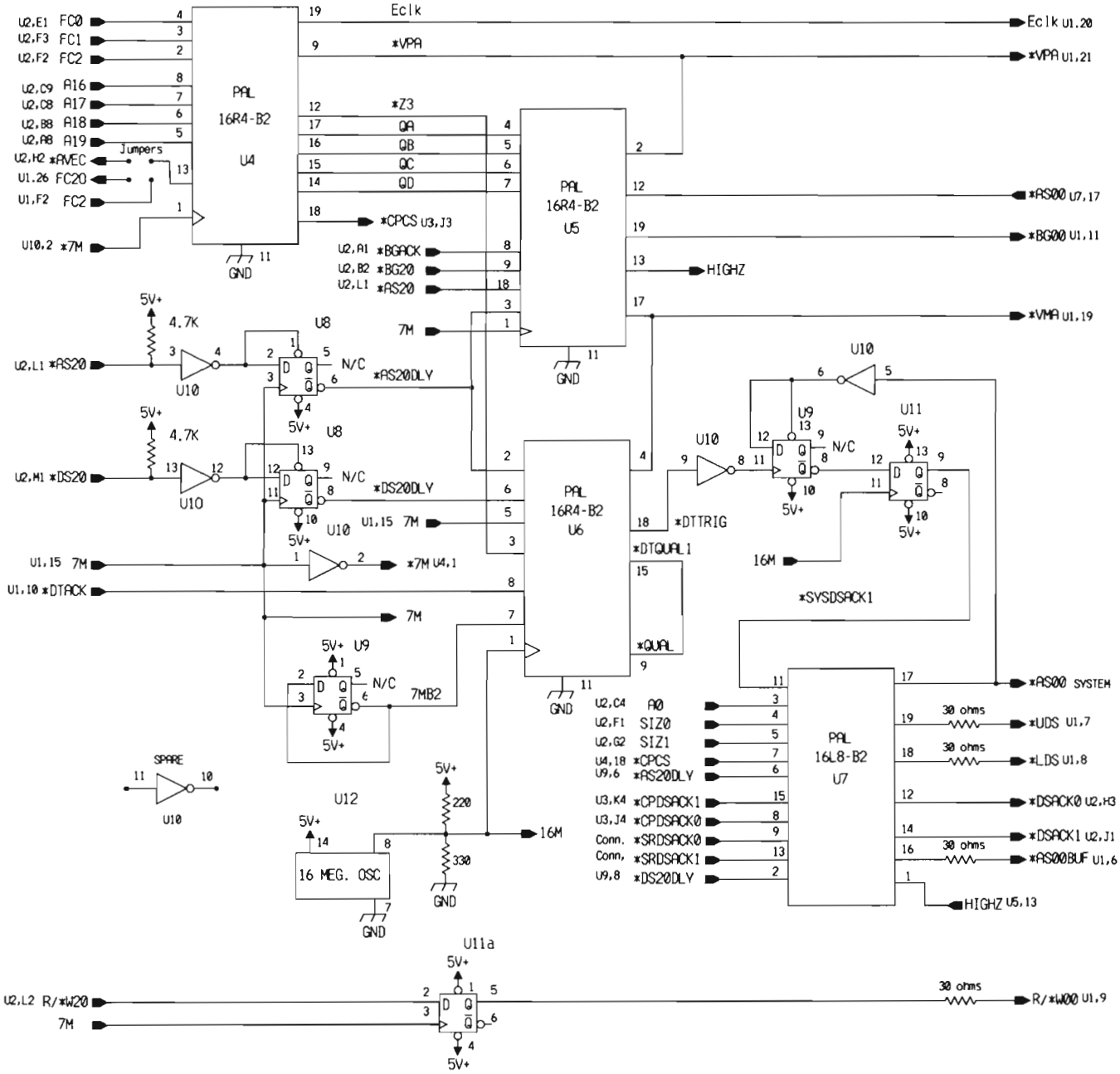
What could be simpler?

*DSACK0 is generated from the 68881 and from the future static RAM only.

## Bus Arbitration

The Bus arbitration technique is quite similar to the 68000 with one exception. During coprocessor cycles the *AS is blocked from the 68000 bus. This gives rise to a possible problem. If the 68020 begins a coprocessor cycle with *AS blocked and responds to an alternate bus master's *BR (Bus Request) with a *BG (Bus Grant), the 68020 will assume the alternate bus master will wait for the negation of *AS. Unfortunately, *AS is blocked and therefore already negated. The result is bus contention. Therefore we must prevent the assertion of *BG until the interface negates *AS. This is done with the equation,

!BG00 = (!BG20) & (!Z2) & (AS20)



**Listing:** Equations for the four PAL chips in CUPL format

```
PARTNO      U4 ;
NAME        Finalu4 ;
REV         03 ;
DATE        May 23rd, 1988 ;
DESIGNER    Brad Fowles ;
COMPANY     Anakin ;
ASSEMBLY    Lucas ;
LOCATION    U4 ;

/* PAL16R4B2 */
/* PAL DESIGN SPECIFICATION  */
/* 68020-68881/68000 AMIGA INTERFACE */

PIN 1 = 7MN ;
PIN 2 = FC2I ;
PIN 3 = FC1 ;
PIN 4 = FC0 ;
PIN 5 = A19 ;
PIN 6 = A18 ;
PIN 7 = A17 ;
PIN 8 = A16 ;
PIN 9 = VPA ;
PIN 12 = Z3 ;
PIN 13 = FC20 ;
PIN 14 = QD ;
PIN 15 = QC ;
PIN 16 = QB ;
PIN 17 = QA ;
PIN 18 = CPCS ;
PIN 19 = E ;

!E  = (!QD )& (!QC )#
      (!QD )& (!QB );

!CPCS  = (FC2I )& (FC1 )& (FC0 )& (!A19 )& (!A18 )& (A17 )& (!A16 );

!FC20   = !FC2I ;

!Z3  = !QD #
       QC #
       QB #
       QA ;

!QA.D = QA ;

!QB.D = (!QB ) & (!QA )#
        ( QB ) & ( QA )#
        ( QD );

!QC.D = (!QC ) & (!QA )#
        (!QC ) & (!QB )#
        ( QC ) & ( QB ) & (QA );

!QD.D = (!QC ) & (!QD )#
        (!QB ) & ( QA )#
        (!QD ) & (!QA );

/*
DESCRIPTION: DECADE COUNTER,6800 CLOCK, AND COPROCESSOR SELECT LOGIC.
*/PARTNO      U5 ;
NAME        Finalu5 ;
REV         04 ;
DATE        May 23rd, 1988 ;
DESIGNER    Brad Fowles ;
COMPANY     Anakin ;
ASSEMBLY    Lucas ;
LOCATION    U5 ;

/* PAL16R4B2 */
/* PAL DESIGN SPECIFICATION  */
/* 68020-68881 /68000 AMIGA INTERFACE */

PIN 1 = 7M ;
PIN 2 = VPA ;
PIN 3 = AS20DLY ;
PIN 4 = QA ;
PIN 5 = QB ;
PIN 6 = QC ;
PIN 7 = QD ;
PIN 8 = BGACK ;
PIN 9 = BG20 ;
PIN 12 = AS00 ;
PIN 13 = HIGHZ ;
PIN 14 = Z2;
PIN 15 = BGACK2 ;
PIN 16 = BGACK1 ;
PIN 17 = VMA ;
PIN 18 = AS20 ;
PIN 19 = BG00 ;


HIGHZ   = (BGACK2 ) & (!AS20DLY )#
          (BGACK2 ) & ( BG20  );


!BG00   = (!BG20 )& (!Z2 )& (AS20 );


!BGACK1.D = !BGACK ;

!BGACK2.D = !BGACK1 ;


!Z2.D     = (!AS00 )#
            ( AS20 );

!VMA.D = (!QD )  & (!QC ) & (QB ) & (QA ) & (!VPA )#
         (!VMA ) & ( QD )#
         (!VMA ) & ( QC );


/*
DESCRIPTION: BUS ARBITRATION LOGIC AND VMA GENERATION
*/PARTNO      U6 ;
NAME        Finalu6 ;
REV         03 ;
DATE        May 23rd, 1988 ;
DESIGNER    Brad Fowles ;
COMPANY     Anakin ;
ASSEMBLY    Lucas ;
LOCATION    U6 ;

/* PAL16R4B2 */
/* PAL DESIGN SPECIFICATION */
/* 68020-68881 /68000 AMIGA INTERFACE */

PIN 1 = 16M ;
PIN 2 = AS20DLY ;
PIN 3 = Z3 ;
PIN 4 = VMA ;
PIN 5 = 7M ;
PIN 6 = DS20DLY ;
PIN 7 = 7MB2 ;
PIN 8 = DTACK ;
PIN 9 = QUAL ;
PIN 12 = DTPRELIM ;
PIN 13 = BUFOUT ;
PIN 14 = DTQUAL ;
PIN 15 = DTQUAL1 ;
PIN 16 = Z1 ;
PIN 18 = DTTRIG ;
PIN 19 = SYSDSPRE1 ;


!DTPRELIM = (!7M ) & ( !7MB2 ) ;

!DTTRIG = (!BUFOUT ) & (!DTPRELIM) ;
```

```
!SYSDSPRE1  = !Z1 #
      (!DTACK ) & (!AS20DLY ) & (!DTQUAL) & (!QUAL) ;


!Z1.D = (!DS20DLY ) & (!Z1 ) #
        (!DS20DLY ) & ( Z3 ) & (!VMA );

!DTQUAL.D = !AS20DLY ;

!DTQUAL1.D = !DTQUAL ;

!BUFOUT = !SYSDSPRE1 ;

/*
DESCRIPTION: SYSTEM DSACK1 GENERATION
*/PARTNO       U7 ;
NAME          Finalu7 ;
REV           05 ;
DATE          May 23rd, 1988 ;
DESIGNER      Brad Fowles ;
COMPANY       Anakin  ;
ASSEMBLY      Lucas ;
LOCATION      U7 ;

/* PAL1618B2 */
/* PAL DESIGN SPECIFICATION */
/* 68020-68881 /68000 AMIGA INTERFACE */

PIN 1 = HIGHZ ;
PIN 2 = DS20DLY ;
PIN 3 = A0 ;
PIN 4 = SIZ0 ;
PIN 5 = SIZ1 ;
PIN 6 = AS20DLY ;
PIN 7 = CPCS ;
PIN 8 = CPDSACK0 ;
PIN 9 = SRDSACK0 ;
PIN 11 = SYSDSACK1 ;
PIN 12 = DSACK0 ;
PIN 13 = SRDSACK1 ;
PIN 14 = DSACK1 ;
PIN 15 = CPDSACK1 ;
PIN 16 = AS00BUF ;
PIN 17 = AS00 ;
PIN 18 = LDS ;
PIN 19 = UDS ;


AS00.OE = HIGHZ ;
!AS00   = (CPCS )& (!AS20DLY );

AS00BUF.OE = HIGHZ ;
!AS00BUF   = (CPCS )& (!AS20DLY );

UDS.OE = HIGHZ ;
!UDS  = (!DS20DLY )& (!A0 ) & (CPCS) ;

LDS.OE = HIGHZ ;
!LDS  = ( !DS20DLY ) & ( SIZ1 ) & (CPCS) #
        ( !DS20DLY ) & (!SIZ0 ) & (CPCS) #
        ( !DS20DLY ) & ( A0 ) & (CPCS) ;

!DSACK1 = (!SRDSACK1 )& (!AS20DLY )#
          (!CPDSACK1 )& (!AS20DLY )#
          (!SYSDSACK1)& (!AS20DLY ) ;

!DSACK0 = (!SRDSACK0 )& (!AS20DLY )#
          (!CPDSACK0 )& (!AS20DLY ) ;

/*
DESCRIPTION: ADDRESS STROBE, UPPER AND LOWER DATA STROBE AND
             FINAL DSACKX GENERATION
*/
```


# ZING!®

**ZING!** - Mouse Driven CLI - $79.95
**ZING!Keys** - Macro program! - $49.95
**The Demonstrator** - Create Tutorials $39.95

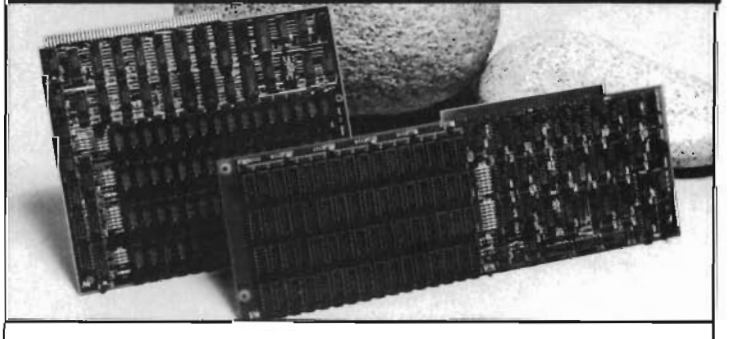
**NEW:**
**ZING!Spell** - Check & Correct your Spelling as you Type; Batch Checking too; ARexx interface; Word expansion; - $79.95
**Cordless Mouse** for the AMIGA - $179.95
**Touchscreen** for the AMIGA - CALL.

Find out about our **Trade-UP** policy which will save you up to **50%**. Write or call today and receive more info on all our products.

MERIDIAN™ SOFTWARE INC.
(303) 979-4140
Credit Cards and Dealer Inquiries Welcome!
9361 W. Brittany Avenue
Littleton, CO 80123

*AMIGA* is a registered trademark of Commodore-AMIGA, Inc.
ZING! is a registered trademark of Meridian Software, Inc.

QUALITY



*8 megabytes for the Amiga!*

ASDG designs and builds in quality from the start. Our 8 meg board comes with 0, 2, 4, 6, or 8 meg installed. Of course it's no-wait-state memory, of course it's fully auto-configuring, and comes in your choice of the Amiga® 1000 or Amiga® 2000 form factor. You wouldn't expect less from ASDG.

We back up our quality with a full 18 month warranty. With memory boards from ½ meg to 8 meg ASDG gives you more choices for memory expansion. For more information contact ASDG at
(608) 273-6585
925 Stewart Street
Madison, Wisconsin 53713

ASDG Inc.
(608) 273-6585

PERFORMANCE

# The Final Arp.Library Tour

## *Spawning child processes and much more*

**by Scott Ballantyne and Charlie Heath**
*Copyright © 1988 Scott Ballantyne and C. Heath*

*Scott Ballantyne is a programmer whose current interests are operating systems and computer language design. He has played a leading role in the ARP development effort since its inception, and wrote much of the arp.library code. Scott may be reached on CIS (70066,603) or BIX (sdb).*

*Charlie Heath, of MicroSmiths Inc., is the author of several well-known programs for the Amiga, notably TxEd, FastFonts and BlitzDisk. He is also the founder and co-ordinator of the ARP project. Charlie may be reached on CIS (74216,2117) or BIX (cheath).*

For those of you who might be kind enough to worry that the title of this article indicates the termination of ARP, let me hasten to assure you that this is not the case - ARP is alive and doing very well, thank you. Our initial goals have been largely reached: it is now possible for users and programmers to work in a largely BCPL-free environment, and programmers now have a significant library of functions that can make life much easier.

The title does refer to the fact that, barring the odd act of God, arp.library is closed to the addition of new functions. With the addition of the functions in the 1.1 release of the library, we feel that most of the bases have been covered as far as providing the common functions most programs would need. We will continue with bug fixes and enhancements to the library as required, but there will be no more additions to the library until many more Amiga owners have more memory.

ARP development is shifting towards new concepts and new programs. We want to let users do things they haven't been able to do before. The kinds of things that are likely to come from ARP in the future are new shells, printer spoolers and Workbench replacements. In fact, I have a preliminary version of the ARP Workbench replacement running on my Amiga, and it looks very impressive and promising indeed. It is being developed by Ken Salmon, who joined the ARP effort for V1.1.

### Acceptance of ARP

We have been very excited and pleased at the acceptance of ARP by users and programmers alike. A lot of commercial and public domain software has already been released using arp.library, and more is on the way. Programs that use arp.library appeal to users because they tend to be smaller and more stable. In my experience, the disk savings of the ARP command programs and programs using arp.library is what users most appreciate about ARP. They just love being able to free up 50K on their Workbench or application disks. While using ARP obviously isn't going to make a bad program usable, ARP can help to make a good program more reliable, and smaller.

A good example of this point can be seen by comparing the FaccII program (ASDG) with BlitzDisk (MicroSmiths). While both programs are excellent, BlitzDisk (which uses arp.library) is considerably smaller, weighing in at 2484 bytes, while FaccII (which does not use arp.library) is 9368 bytes in size. The run-time overhead is significant as well: BlitzDisk uses about 2K of overhead, while FaccII uses about 19K - more RAM than both BlitzDisk and arp.library occupy together.

Performance is also usually improved by the use of ARP. The standard Amiga Workbench causes much grinding and gnashing of the Amiga's disk drives as it searchs for .info files, and the various other file system objects. The ARP Workbench replacement starts quickly and quietly, with no muss and no fuss, largely because it uses the ARP functions, which are efficient and well behaved, for searching disks and device lists.

The acceptance of ARP by Commodore is a matter of current negotiation, so I guess I can't say much about it here. I can say that we are still interested in letting CBM have ARP, and they appear to be interested in getting it. If you are a user of ARP, you might want to send a letter to Max Toy or Gail Wellington at Commodore's West Chester headquarters and tell them how much you like it.

Of course, even if ARP doesn't end up on a future Workbench disk, it appears that most of the goals of ARP will still be realized. Much of the 1.3 software (still yet to be released at this writing) by Commodore contains improvements first found in ARP, although they lag behind ARP in some important areas. Hopefully, the 1.4 release will see either ARP itself, or the ARP goals realized even more completely, with genuine process

management and sensible language interfaces. However, a conservative estimate would place a 1.4 release a year, or more likely, two years away. Which means that real process control, not to mention many of the other improvements ARP brings to the Amiga, are a good distance down the road.

In sum, ARP, by its wide acceptance among users and programmers alike, is a message to Commodore that things need fixing. Whether they use our solutions, or redo our work in-house doesn't matter in the grand scheme of things. ARP started out as a project to improve the Amiga, and whether it remains a spur to improvement or becomes an official part of the Amiga software release, it will have reached its most important goal.

**So what's new in ARP 1.1?**

Aside from the bug fixes, GADS() (the command line argument parser) and especially FileRequest() have been enhanced. GADS() now supports a special new template type that allows you to enter as many arguments as you want, with no more artificial limits and no more ugly strings of commas in the templates. FileRequest() has been dramatically enhanced, so much so that it deserves a place in this article along with the new functions we will talk about.

There are now date conversion routines in arp.library that support international date formats for input and output, giving ARP and the Amiga a truly international flavour.

For my money, the most exciting new addition to arp.library is the ability to spawn child processes. You can start a separate process from within your program, then go about your business. You don't need to perform any clean-up at all - the child process will take care of itself, which means the parent can even terminate before the child. This has never been possible before on the Amiga; it is possible now only with arp.library.

We have also added support for Resident programs, and have put forward a proposal for a resident standard that works with 'dirty' data as well as 'pure' data. There are other approaches to resident programs out there, and I will talk about the pros and cons of each.

**Process management under AmigaDOS**

Before demonstrating ARP's solution to the problem of process management on the Amiga, a little background on the 'native' facilities is appropriate. As you probably know, there is a distinction on the Amiga between Exec processes and DOS processes. You are supposed to use the word 'task' to refer to Exec processes, and 'process' to refer to DOS processes, but in practice this often doesn't happen, and it probably won't happen here. I will usually be speaking about DOS processes - if I do refer to 'tasks', I will say something like 'Exec task'.

There is also an unfortunate distinction that must be made between Workbench processes and CLI processes. The difference I refer to is not the user interface, but the lack of uniformity and integration between the CLI and Workbench environments from the point of view of the program. Input and Output are problematic and inconsistent between the two environments, and stack-checking techniques that work for CLI do not work at all for Workbench processes.

AmigaDOS provides two functions for creating processes, CreateProc() and Execute(), and each has its own specific drawbacks. CreateProc() can be used to create an asynchronous process, and actually works well within the limitations of its design. The main problem with CreateProc() is that it does not provide for any automatic process clean-up when the child it has created terminates - you must hang around to unload the program when it is done. A related problem is that CreateProc() will also not inform you when the child has terminated - you must get a message from the child when it dies or commit that ultimate sin against multitasking and poll a system list to see if it is still around. This limits the use of CreateProc() to programs with which you have made prior arrangements. It is not possible to establish a safe input/output environment for programs, and the stack size is found in a different location from that of CLI children.

CreateProc() is the function that Workbench calls to launch programs, and it is largely responsible for the silly and arbitrary distinction between Workbench programs and CLI programs that exists on the Amiga system.

To run CLI programs, you must call Execute(). Execute() will create CLI processes, but they are not asynchronous - your process (the parent) will sleep until Execute() fails, or until the child terminates. It is also impossible to get a valid return from Execute() - the sad truth is that you will never know for sure if the programs you were trying to Execute() actually ran. This is because Execute() actually loads the Run program from the C directory, which does the actual work of spawning the process. If Execute() can't find Run, it will return a 0 to you, indicating failure, but if Run can't find your program, or it fails for any reason, you will never find out about it. And if you actually do try to spawn a background CLI process using a command like "Execute("RUN Type File to PRT:", 0L, 0L);", you are in double trouble - since you get to run Run twice. They say all madness has a simple beginning, and it must be true, since that simple line of code is what started my involvement in ARP. I logged onto BIX to flame about AmigaDOS, discovered Charlie Heath's project, and signed up.

This only scratches the surface of the problems you can have with Execute(). The following pair of programs illustrates the difficulties caused by the inconsistent handling of input and output between Workbench and the CLI. The first program simply writes a little message to the current output stream:

```
main()  /* ProblemChild.c */
{
    long Write(), Output();
    Write(Output(), "Hello World!\n", 13L);
}
```

The second program loads and runs the first program.

```
main() /* ProblemParent.c */
{
    long Execute();
    Execute("RAM:ProblemChild", 0L, 0L);
}
```

Compile these two programs, and make an Icon for ProblemParent (copying the Clock.info file as ProblemParent.info should do fine). Then double-click on the ProblemParent icon. At this point one of three things can happen:

1) You could crash.
2) You could hang (it will appear that nothing happens).
3) You could see the message "Hello World!" in the CLI that you used to launch Workbench.

If you get #3, close the CLI using the EndCLI command, then double-click the icon a few more times. In most cases, nothing will appear to happen. Open up another CLI, and do a Status - surprise!, you will notice that all of a sudden you have many background CLIs, and they all say "No Command Loaded". The only way to recover the resources associated with these CLIs is to reboot.

You might decide to use the other arguments to Execute() to actually set the input and output file handles. This has its own set of problems; one of the strangest is illustrated by this program:

```
main()  /* Trouble.c */
{
    long Execute(), Input(), Output();
    Execute("Type Trouble.c", Input(), Output() );
}
```

To see the problem, try entering a command line such as the following:

```
1> Trouble <Trouble.c
```

You will eventually end up with the mysterious message:

```
Unknown command main()
```

This is because Execute() insists on interpreting a non-null input stream as a series of commands! And of course, all of this assumes that the parent is still around, managing those file handles for the child. You can completely forget about setting input and output file handles for independent processes.

There is a final wonderfully irritating problem with Execute() - there is no way for you to have a child process respond to a CTRL-c interrupt from the keyboard. In fact, in most cases, the CTRL-c will be sent to the parent, so if you use Execute(), there is no way to tell if the user that sent the CTRL-c intended to interrupt you (the parent) or the child process!

**An integrated environment**

One of the nicest aspects of the ARP process control functions is that they are fully integrated with other related aspects of the system. For example, both ASyncRun() and SyncRun() search the ARP resident list, and then the command PATH set by the CLI when they look for the command to execute. This allows shells built on top of arp.library to work well with all user programs and features. For example, the ARP Workbench replacement will automatically search the resident list and the CLI Path for programs, unlike the current Workbench. (Incidentally, the ARP Workbench will also be able to tell when you do an Assign from the CLI.)

ARP also builds a consistent stack frame that can be used for stack checking. The frame used is the one that the AmigaDOS Developers' Manual documents (but AmigaDOS only occasionally uses). The Input/Output problem required a different approach. You can specify a default case that will select the 'normal' I/O handles for each environment, or you can specify an alternative that will always create a standard input/output environment, running the process in its own window, and allowing it to respond to CTRL-C interrupts.

You are not tied to either of these - you may set one or more of the file handles for the child yourself, and even run the child with NULL input or output handles if you want. Finally, while an asynchronous CLI process is the default process type created (and this is the only way, currently, that you can create such a process), you can elect to create a Workbench process as well and even make pieces of your program into processes!

**ASyncRun()**

The function call you use to make all this happen is ASyncRun(). ASyncRun() takes three arguments: the name of the command to run, the arguments you want to pass to the child, and a pointer to a structure called a ProcessControlBlock. ASyncRun() will often supply its own default values for NULL entries in the ProcessControlBlock, so it is generally quite important that you set all unused variables to NULL.

(Before continuing, I should report a bug that crept into the arpbase.h header file in the very first release of ARP. In these files, there are two pcb_SplatFile variable declarations; there should be only one. The second one should be deleted, leaving the one declared inside the pcb_Console union. More recent releases (such as the one for *Transactor for the Amiga*) do not have this problem.)

The simplest way to use ASyncRun() is illustrated in the program Defaults.c, which appears at the end of this article as Listing 1. (Incidentally, you will find it instructive to run a few of these examples from both the CLI and the Workbench.) Defaults.c sets only the process stack size and priority (which are always required), and defaults to ASyncRun()'s judgement for the rest of the structure. If you are running from CLI, a background CLI process is created, very much like the one created

by using the AmigaDOS Run command. The child will share the current output window with the parent, and use the NIL: device for input. The only way to signal the task is to use the Break command. If you run from Workbench, ASyncRun() will set the PRB_STDIO control bit (see below) for you, and open up a console window for the child process.

If you want the process to be an interactive process, and to run in its own window, the easiest approach is to set the PRB_STDIO bit in the pcb_Control variable. This causes the same behavior from both CLI and Workbench, which is the creation of a complete standard I/O environment, and a console window that will pass CTRL-C signals to the correct process. An example of how to do this is contained in the simple program stdio.c (see Listing 2).

**Setting your own input/output handles**

ASyncRun() will also allow you to open your own input and/or output files for the child, and will automatically close them for you when the child terminates. If you want to do this, then you must not set the PRB_STDIO bit, since that causes ASyncRun() to ignore the values in pcb_Input and pcb_Output. You do not need to supply both input and output handles; ASyncRun() will take care of the one you omit to the best of its ability.

One possible problem you can have when you take a little more control like this is the unfortunate case of the Workbench child who does an Open("*", MODE_mumble). This will cause the child to hang - the Open() call will never return. ASyncRun() cannot do anything about this on its own, but it can do something about it with your help. If you have a valid CON: window opened, you can pass its handle to ASyncRun() as follows:

```
struct ProcessControlBlock pcb;
pcb.pcb_Console.pcb_SplatFile = Console_FileHandle;
```

ASyncRun() will arrange things so that if the child does an Open("*"), it will use the console file in pcb_SplatFile. You should note that this file will not be automatically closed by ARP when the child exits - if you do want ARP to close it, you must set the control bit PRB_CLOSESPLAT in the pcb_Control variable and, of course, in this case you must not close the file yourself or assume it is open for your use. Note that this care is only necessary when not using the defaults *and* running as a child of a Workbench process.

Finally, there are cases when you want total control of the child's handles. If so, you can set the PRB_SAVEIO bit in the pcb_Control variable, and ASyncRun() will let anything through, and will not close the files on exit. Using this bit, you can also pass NULL file handles to the child, which has proven useful in certain cases. Note that this option is independent of the pcb_SplatFile and PRB_CLOSESPLAT options discussed earlier - it is possible to use both PRB_SAVEIO and also to set a default console window.

**Process synchronization**

While running independent processes is what ASyncRun() is all about, it is often convenient to find out when the processes are done. Using ASyncRun() you can spawn a few tasks, do some processing of your own, and then wait at a convenient point for the sub-tasks to finish. ASyncRun() can be asked to send you a message when the child process terminates. The message sent is called a ZombieMsg, and contains several useful tidbits of information, such as the process's exit status, the system's status at time of exit, and the time of exit. There is also room for you to attach your special software structures. The coordination of sub-processes is illustrated in sync.c (see Listing 3).

As shown in sync.c, you need only obtain a MsgPort, initialize the ReplyPort in the ExecMessage portion of the ZombieMsg to this port, launch the process, and wait for a reply. Although we used two PCB structures for convenience and clarity, there is actually no need to do this. ASyncRun() does not use the ProcessControlBlock after it returns to you, so it may be reused as desired. On the other hand, you obviously need as many ZombieMsg's as you have processes, since that is owned by the child until it is returned to you.

The portion of the code where we wait for the ZombieMsg deserves some comment, as does the clean-up portion of the program. Normally, ASyncRun() will return the CLI process number of the child to you. However, if there is a problem, the return from ASyncRun() will be one of the negative error codes documented in the ARP header files. What happens in the case of an error is important - ARP will clean up as though the child ran normally. That is, unless you specifically override this behaviour, all open files will be closed, the code unloaded, and so on. However, since the process is dead on the return from ASyncRun() in a case like this, you do not need to get a ZombieMsg, so it will not be sent. It actually all reduces to these simple rules:

- Never clean up after a task, unless you have told ASyncRun() you want to.
- Only wait for a requested ZombieMsg when you have a positive return code from ASyncRun().

The rest of the program should be pretty self-explanatory. We use a custom software structure that we attach to the ZombieMsg for each task to keep track of the command name, departure time and ASyncRun() return code. This can be extended or changed as you wish, since ASyncRun() completely ignores the UserInfo variable. We also use arp.library's datestamp-to-string conversion routines (more on these later) to convert the date stamps to human readable form.

**Alternative lifestyles for processes**

ASyncRun() creates a CLI process by default. However, if you need to do a CreateProc() style of process, that is available to you by setting the PRB_NOCLI control bit. Of course, this

involves you in more complexities, since most programs out there will either be expecting you to send them a Workbench start-up message, or won't work properly from this environment. On the other hand, ASyncRun() will still handle most of the clean-up work for you, including unloading the program code and closing file handles, so if you have to do CreateProc() things, using ASyncRun() to do them is probably best.

Ordinarily, ASyncRun() returns the CLI number of the child on a successful launch. However, there is no CLI number associated with non-CLI processes, so ASyncRun() returns a zero if you use PRB_NOCLI, and there is not an error. It also returns with a pointer to the child process's MsgPort in the pcb_WBProcess variable in the ProcessControlBlock structure, which you may use to communicate with the non-CLI process you have just created.

As an example of its use, wbproc.c (Listing 4) loads and runs the standard Clock program with PRB_NOCLI set. Note that the parent task must manufacture and send a Workbench-style start-up message, and must also wait for the reply.

Although we didn't have to, we chose to load and unload the program ourselves, using the ARP function calls LoadPrg() and UnLoadPrg(). These are the calls ASyncRun() uses to find the command that you supply as its first argument. They properly handle the resident list and path, and return the same values as the AmigaDOS LoadSeg() call, namely a BPTR to a seglist, or NULL, if there was an error.

If you already have a program 'in hand', so to speak, you can tell ASyncRun() where it is by installing the BPTR to its seglist in the pcb_LoadedCode variable in the ProcessControlBlock structure. This allows you to run multiple copies of the same code at the same time, should you be so inclined. If you do use this feature, you must hang around to unload the code when the child has finished with it.

Note that although wbproc.c doesn't illustrate it, even CreateProc()-style processes can have their input/output file handles set, and will use them if the program does any kind of standard I/O. As usual, ARP will close them for you on exit.

**Strange and wonderful?**

Finally, ASyncRun() lets you turn pieces of your program into processes. This is a common technique with the lower level Exec tasks, but it can't easily be done with DOS processes due to AmigaDOS's requirements for BPTRs and seglists and the rest.

To do this, you install a CPTR (note: NOT A BPTR!) to the start of your code in pcb_LoadedCode, and also set the PRB_CODE bit in the Control variable. That's pretty much it. Of course, if you do this, you can't exit before the child - remember, you were loaded together. Aside from this, you can do everything an Exec task can do, and you can also do everything a DOS process can do, including open files and libraries, send output to printers, the whole enchilada, and with no speed penalty.

An example of this is shown in the program magic.c (Listing 5). As a quick glance at the program will show, it is extremely simple to use and to set up such a sub-process. The only really new details involve setting up the environment for the sub-process. Aztec C users who use the small model must call the geta4() function when the process is first started up, to initialize the compiler's pointer to its data area. Lattice C users who are using the small model must specify the -y switch, which does the same thing.

One important point to remember when doing this sort of thing with ASyncRun() is that you must be very careful about calling any compiler library functions. That's why MyProcess() exits with a capital E, rather than the small exit() that is in the compiler library. The problem here is that C programs typically start with some compiler-specific code that allocates file buffers and so forth. You are not entering through this code when you start up MyProcess(), so you don't have the compiler environment. Unless you want to set it up yourself, stick to operating system calls.

**Resident programs, the what and the why**

Typically, on the Amiga, when you run more than one instance of the same program, you have more than one copy of that program's code in memory. Resident programs are programs that need only one copy of the program to be in memory, no matter how many simultaneous invocations of that program there are at any given time. This can provide a large memory savings, and can also save you time waiting for programs to load. Obviously, this is very appealing, particularly for programs such as text editors, or often used commands like Dir. Probably because of this very appeal, there are several different approaches to doing resident programs on the Amiga. To evaluate and understand these different approaches, it is important to understand the issues and problems involved in running resident programs.

**The criteria for resident programs**

Since more than one process might be using the same copy of the code at the same time, it is obviously important that the code segment does not modify itself. Most Amiga programs will fit this criterion, although there are a growing number of programs that do something called 'SegList Splitting', which is the binary equivalent of chopping off your own head, especially with regard to becoming residentable.

The same thing is true for data - since more than one process could be reading the same data at the same time, it is important that no static data be altered. Data that is changed must be kept in registers, in allocated memory, or on the stack. Most Amiga programs will *not* meet this criterion, and will behave unpredictably or crash as resident programs. Programs that do meet this second criterion are referred to as 'pure code and

data' programs, or simply as 'pure'. There are not many of them; the original BCPL command programs are pure programs, as are the ARP replacements. There are probably other 'pure' programs around, but they are certainly not plentiful. By the way, do not use code purity as a criterion of program excellence - modifying static data is perfectly acceptable, even to the software priesthood.

We are now in a position to talk about the different approaches to Resident programs, including ARP's. Each approach falls basically into two groups. Group one will only run pure code programs, while group two attempts, in some fashion, to allow impure programs to be made resident as well, through some data cloning capability built into the program. Those in group one are all commercial shell programs, the original, released only to developers 1.2 resident CLI, and from all reports the forthcoming 1.3 CLI. Group two currently has only two members: ARP, and another freely distributable program called REZ, by software wizard Jim Goodnow II, author of the Manx C compiler for the Amiga.

The programs in group one are of limited use. There are many programs (such as text editors) that it should be possible to make resident which group one programs cannot handle. Some of the commercial shells for the Amiga that implement resident programs do not even check to see if the program is pure or not. This can lead to instant crashes, and is not very pleasant or helpful.

**ARP Resident and REZ**

There are several similarities between ARP's Resident and the REZ program. Both handle pure code programs, and both will usually be able to run only small-model higher level language programs. Each provides a data cloning facility for impure programs, and there is user protection in the form of checksums to detect dirty code or perhaps, damage caused by an errant program.

They differ largely in details and approach. Goodnow's approach is to SetFunction() the LoadSeg() and UnLoadSeg() library calls. When a program that the user desires to make resident is loaded, the relocation information for that program is saved in an Exec-style library called REZlib.library, and the program code segment is patched to point to a special geta4()-style routine in REZlib.library. This is a very clever approach with a lot to recommend it. Its main advantage is that it will allow much more 'off the shelf' software to be made Resident without recompiling than the ARP Resident approach will. The biggest disadvantage is that it has a much higher overhead than the ARP resident.

ARP's current approach to Resident programs is to provide a special Resident header at the start of a program. This header, aside from identifying the program as one that can be made resident, also records the size of the stack and data areas required by the program. This header is recognized by the arp.library program loaders and process managers, which cooperate to allocate the data and stack as indicated in the Resident header. This allocated data is passed to the program in the a4 register. Note that ARP will not refuse to load programs without this header but, unless they are pure, you will not succeed in running them as resident programs more than once. Note that ARP always checksums programs before running them to prevent crashes (this is a user-settable option in REZ).

So which is better? That is an impossible question to answer, and I use both. REZ handles programs written before ARP and the Resident standard were formulated, while ARP provides dramatically improved memory savings with programs that support the Resident standard. ARP requires a slight amount of additional work on the part of programmers to cooperate with the ARP standard, while REZ typically does not. On the other hand, there are programs that REZ cannot run as shared processes, including Manx's own Make utility and compiler, so having the developer of a program take the small amount of work necessary to make a program work as resident might not be a bad idea.

**Making a program resident**

Since programmers must make special efforts to make a program residentable under ARP, a natural question is exactly how much effort it takes, and how careful do you have to be. The answer is, not much and not very, respectively. I use an Intuitionized version of the MicroEmacs editor for a lot of my Amiga hacking, and I decided to convert it to a resident program as a test case. The total time I spent on this was under an hour. I now save 50K each time I use the editor, which is a nice savings, especially at today's memory prices.

So what do you have to be careful of when you write a program that you would eventually like to have Resident? Probably the most common difficulty is with static data initializations such as the following:

```
struct Foo
{
    struct Foo *next;
    ULONG dummy;
};

struct Foo foo_array[] =
{
    {
        &foo_array[1],      /* This is the problem */
        0,
    },
    {
        &foo_array[2],
        1,
    },
};
```

This is a fairly common occurrence in programs, especially on the Amiga, which uses a lot of structures (such as Menus) that

must be linked together. The problem with such an initialization is only when the structures are written to; if they are read only, you don't have to worry. However, if a program is made Resident, and the structures are modified at run-time, the original of the data is modified, not the copy made by ARP for the program to use. If this happens, ARP will detect it, and put up a requester informing you of the difficulty.

There is a solution to this problem, and it is quite simple: just initialize all compiled references from one data item to another data item in a simple loop at run time. Your program will then run correctly.

Something to watch out for that will cause problems for both REZ and ARP's Resident was discovered by Jim Goodnow, and goes like this:

```
char *bra;

bra = "bxx";
strcpy(bra+1, branches[index]);
```

The problem here is that the compiler stores strings declared in this fashion in the code segment, so the code segment gets modified by the strcpy() call. The solution is quite simple:

```
char bra[] = "bxx";
```

**The ultimate overhead calculation**

So now that you have seen how really easy it is to get things to work as Resident programs, what is the final cost of the ARP Resident versus REZ? Taking the MicroEmacs editor I mentioned earlier as an example, we have a program size of about 32K, including both data and code. REZ must save a certain amount of extra relocation info for each program that ARP does not need to save, but the real savings comes in the hidden overhead of the system stack allocation. People typically put the system stack at around 20K, which means that every program you run uses that much memory. Obviously, this is often far more RAM than is required. The MicroEmacs editor requires a stack of only 4K, so the ARP resident program saves 16K over REZ in this instance for each invocation. That adds up to a lot of memory savings.

So which is better? As I mentioned before, I use both. New programs are better advised to use the ARP Resident header due to its lower overhead. They work well together, and I think we should look at adding some of REZ's backward compatibility to the ARP standard. We could save the relocation information for programs that lack the resident header, and still get the advantage of the memory savings with newer programs.

**Time for ARP to go international**

There are two date conversion routines in this release of ARP library. One, StampToStr(), takes a DateStamp and converts it to a human-readable string, while StrToStamp() converts a human-readable string to a DateStamp structure. Both of these routines can convert to and from one of four date formats: International, USA, Canadian, and the current system AmigaDOS default. To select one of these formats, we have adopted the use of an environment variable called 'dateformat'. If this variable is not defined, the current AmigaDOS-style display is used, otherwise, one of the following is selected:

| Value of dateformat | Format Selected | |
|---|---|---|
| 0 | AmigaDOS, | dd-mmm-yy |
| 1 | International, | yy-mm-dd |
| 2 | USA, | mm-dd-yy |
| 3 | Canadian, | dd-mm-yy |

Any illegal value will cause a default to the AmigaDOS style of display. If you precede any of the above values with a '-', then the display of names like 'Monday', 'Tuesday', etc., in commands like List will be suppressed. All of the ARP commands that use dates (List, Date, SetDate, for example) use the dateformat and these values to control the display and input format of dates. The ARP conversion functions do not read the dateformat variable for you, since that would force a conversion that might not always be convenient, so you must examine the value of dateformat for yourself to present a consistent interface to the users.

The good news is that it is very simple to do this, and the canonical approach is shown in the sample program DoDate.c (Listing 6). Note that we have to check explicitly for the minus sign in the dateformat string, and not rely on a negative number return from Atol(), since the user could do:

```
set dateformat=-0
```

to indicate the AmigaDOS format with no substitutions, and we would miss it since the zero value is always returned as positive. The only other point worth mentioning is that the two flag values DTB_SUBST and DTB_FUTURE affect only StamptoStr() and StrtoStamp() respectively. This means it is not necessary, for example, to clear DTB_FUTURE before calling StamptoStr().

**The ARP FileRequester, part II**

I like to keep things simple, functional and clean, and this is the philosophy behind the ARP file requester. The basic requester provides a consistent and functional way for applications to get input from the user, and with V1.1 of ARP it is possible for programs to customize the requester for their particular needs.

Most of the file requester's capabilities are pretty obvious to the user, once they've seen it a couple times, but there are a couple of features that aren't so obvious unless you know about them. The first is the capability of displaying a list of devices available (such as df0:, df1:, dh0: and ram:). Normally

the file requester will only display the files in the selected directory, but by clicking the *right* mouse button the display also includes available devices and assigned directories. Most other file requesters that provide this capability do it by displaying icons for the various disk drives, but that often leads to a display that is either unwieldy or incomplete.

The second less obvious feature is the capability of limiting the displayed files to those matching a pattern. If you enter a file pattern at the end of the Drawer name, the displayed files will only be those files which match that pattern. This can be handy when you're searching for a particular file in a directory with a lot of different files. The pattern matching uses the ARP PatternMatch call, which allows all of the standard AmigaDOS wildcards plus the '*' pattern.

For programmers, using the ARP file requester is as simple as defining the input data structure, and making a function call. There are some more complicated capabilities you can use, which will allow you to customize the file requester for your particular program's needs, but first let's go through the basics. The following lines of C code are all you'll need for the simplest case of using the file requester:

```
/* Define these data structures in your program:          */
char  FileName[FCHARS+1];
char  DirName[DSIZE+1];
struct FileRequest MyFileRequest =
{
   "My Prompt String",/* Hailing Prompt                   */
   FileName,          /* Pointer to file name buffer      */
   DirName,           /* Pointer to dir name buffer       */
   0L,                /* Calling window or NULL           */
   0,0                /* Flags for special control (none) */
   0L                 /* Special function (none here)     */
   0L                 /* Reserved                         */
};
...
/*    Then, within the program, use something like the    */
/* following function call go make the file request:       */
if ( FileRequest( &MyFileRequest ) == 0L )
{
Printf("User Cancelled!\n");
} else
{
Printf("User entered file name: %s directory: %s\n",
FileName, DirName);
}
```

In this example, the two buffers 'FileName' and 'DirName' will get the file and directory names the user selects in the file requester. If the user selects CANCEL, the FileRequest function will return ZERO, but if the user selects OK or presses RETURN within the filename gadget, the return value from FileRequest will be the pointer to the FileName buffer. (Note that, if the user selects the CANCEL gadget, the buffers FileName and DirName may still have been changed from their original contents by the user.)

The first addition you may want to make within your program is to set up the fr_Window input value for the file requester to match your particular Intuition Window. This is useful for two things. First, if your window is in a custom Screen, the file requester will open up on that screen if you have initialized fr_Window. If you haven't, the requester will open up on the Workbench screen, which can be rather confusing for your user who is still staring at your beautiful custom screen and contemplating a call to customer support. Also, when you initialize the fr_Window value, the file requester will use the same IntuiMessage IDCMP port as your calling window (unless you set the FRF_NewIDCMP bit in the fr_Flags field). This will simplify handling of events, and also is needed if you want to set up a special function for handling other events within your window while using the file requester (by setting the FRF_DoMsgFunc bit).

Another simple addition you can make is to set the FRF_DoColor bit in the fr_Flags value; this will let you select the second colour set for use with the file requester. By convention, the second colour set will normally be used when you are doing a file save operation, whereas the normal colours are used for a file open operation; this lets the user know at a glance which operation has been selected.

You may also want to use some of the special customizing capabilities in the file requester, which can be set up by initializing the fr_Flags bits desired, and the fr_Function field. The fr_Function field is a *pointer to* a function that will be called under various conditions that you specify by setting the corresponding bits in the fr_Flags field.

The user-supplied function will be called whenever one of the requested conditions occurs. When the function is called, you can tell what particular condition caused the call by checking the 'bits' value, as shown in this example:

```
My_fr_Function(bits, funcvalue)
long   bits;
long   funcvalue;
{ switch ( bits )
   {
      case FRB_NewWindFunc:
         dosomestuff();
         break;
      case FRB_AddGadFunc:
         domorestuff();
         break;
      ...
   }
}
```

The customizing capabilities you can add include:

FRF_NewWindFunc lets you change the NewWindow structure that will open the file requester window. You might want to change the window's position or size, for example, with this function. Changes are at the discretion of the programmer, but

be warned that if you do something weird, a hex may be placed on your program! Here, the funcvalue variable is a pointer to the NewWindow data structure.

FRF_AddGadFunc lets you add your own customized gadgets to the file requester window. This function is called *after* the window is opened, but *before* any of the standard gadgets have been displayed. Any gadgets you add should be ready to be displayed, and positioned where you want them in the window (you've probably already made the window bigger with the FRF_NewWindFunc call-out). Here, the funcvalue variable is a pointer to the actual Window of the file requester. Note, all of your gadgets should have a GadgetID value of less than 0x7680, or greater than about 0x7700.

The FRF_GEventFunc call lets you process the special gadget events which you have added using FRF_AddGadFunc. funcvalue contains the GadgetID for the selected gadget.

If you set FRF_DoWildFunc, you can set up your own special 'filter' function for file names using whatever wildcard code you want to use. Here, the calling value funcvalue is a pointer to a FileInfoBlock that has the requested file name; you should return a value of TRUE if you want to display this file name, or FALSE if you want the name *not* to be displayed. Actually, that's how it is *supposed* to work, but in the current release of ARP.library there is a bug that causes your return value from this and the FRF_GEventFunc to be ignored. Bug fixes to the library are on the way and will be released shortly.

The FRF_DoWildFunc bit will let you process IDCMP messages that are received for the calling window, or whatever other windows are sharing the same IDCMP port. Whenever an IntuiMessage comes into the shared IDCMP that is not addressed to the file requester window, the fr_Function will be called, with funcvalue being a pointer to the IntuiMessage. Note that your program is responsible for replying to this IntuiMessage, if you have set up FRF_DoWindFunc!

Most of this stuff is illustrated in the Freq.c program (Listing 7), which shows first the simplest possible use of the file requester, then opens a file requester in a custom, lo-res screen. It also shows how to add a custom gadget, and a few other things you might find interesting and useful. Note that some of these things will only become fully functional when the new bug fixes for arp.library are released.

That's about it for the special capabilities... if there is some extension you'd like to see, drop us a note at ARP support and we'll consider it for the next revision. If you're using the special customizing functions, you'll probably need to look at the ARP Programmers' Documentation, which contains a somewhat more definitive technical description of the special cases.

### That's all folks!

That's it for now. All of the folks who have worked on ARP want to thank you all once again for the great support, bug reports, and suggestions we have received. More than most software projects, ARP has grown to involve a huge portion of the Amiga user community, and we hope this will continue in the future. So keep those suggestions coming in, and if you have any really nifty programs that you would like to contribute, why not get in touch with us? Our address is:

ARP Support
c/o Microsmiths, Inc.
PO Box 561
Cambridge, MA 02140

## ARP Programming Examples

### Listing 1: defaults.c - Simple ASyncRun() demo

```
/* defaults.c -- Copyright (c) 1988 by Scott Ballantyne
 *               Use and abuse as you please.
 *
 * Use *all* default settings for ASyncRun().
 *
 * compile normally, then:
 * MANX:
 * ln defaults.o -larp -lc
 * Lattice:
 * blink arpc.o defaults.o to defaults lib lib:arp.lib lib:c.lib lib:amiga.lib
 */

#include <exec/types.h>
#include <libraries/arpbase.h>
#include <arpfunctions.h>

struct ProcessControlBlock PCB =
{
   20000,        /* StackSize */
   0,            /* Pri */
   /* Rest get initialized to ZERO by C compiler, which is what we want */
};

VOID
   main()
{
   LONG  rc;

   if ( (rc = ASyncRun("Type", "Defaults.c", &PCB)) < 0 )
      Printf("ERROR CODE = %ld\n", rc);
   else
      Printf("Process Number = %ld\n", rc);
}
```

### Listing 2: stdio.c - Create an interactive process

```
/* stdio.c -- Copyright (c) 1988 by Scott Ballantyne
 *            Use and abuse as you please.
 *
 * How to setup a standard input/output CLI environment for all
 * cases, whether parent is a Workbench process or a CLI process.
 *
 * compile normally, then:
 * MANX:
 * ln stdio.o -larp -lc
 *
 * Lattice:
 * blink arpc.o stdio.o to stdio lib lib:arp.lib lib:c.lib lib:amiga.lib
 */

#include <exec/types.h>
```

```
#include <libraries/arpbase.h>
#include <arpfunctions.h>

struct ProcessControlBlock PCB =
{
   20000,        /* StackSize */
   0,            /* Pri       */
   /* Rest get initialized to ZERO by C compiler, which is what we want */
};

VOID
   main()
{
   LONG  rc;

   /* Setup for STDIO using Console window of our choice */

   PCB.pcb_Control |= PRF_STDIO;
   PCB.pcb_Console.pcb_ConName = "CON:0/0/320/200/STDIO Window!";

   if ( (rc = ASyncRun("Type", "?", &PCB)) < 0 )
      Printf("ERROR CODE = %ld\n", rc);
   else
      Printf("Process Number = %ld\n", rc);
}
```

## Listing 3: sync.c - Co-ordinating sub-processes

```
/* sync.c -- ARP demo program for processes (ASyncRun()).
 *          Illustrates:
 *             Process synchronization using ZombieMsg's.
 *             PRB_SAVEIO control bit
 *             Setting output file handles for children.
 *             Misc other stuff.
 *
 *     Copyright (c) 1988 by Scott Ballantyne
 *     Use and abuse as you please.
 */

#include <exec/types.h>
#include <libraries/arpbase.h>
#include <arpfunctions.h>

/* The PCB for status */

struct ProcessControlBlock Status_PCB =
{
   10240,
   1,                    /* We want this task to have priority */
};

/* The PCB for Type */

struct ProcessControlBlock Type_PCB =
{
   10240,
   -1,           /* We want this task to boogie a little less quickly */
};


/* The messages we will be sent when our children die - initialized in
 * the body of the program.
 */
struct ZombieMsg Status_Zombie, Type_Zombie;

/* A structure we attach to the Zombie message to give us a little more
 * information.
 */

struct MyUserInfo
{
   LONG  Return;                /* Initial return from ASyncRun() */
   BYTE  *Name;                 /* Command name                   */
   struct DateStamp StartTime;  /* Starting time (approx)         */
} StatusInfo, TypeInfo;

/* Date and time conversion stuff */

BYTE  Day[LEN_DATSTRING], Date[LEN_DATSTRING], Time[LEN_DATSTRING];

struct DateTime dt =
{
   0,0,0,                /* The datestamp */
   FORMAT_DOS,           /* For the moment */
   0,
   Day,
   Date,
   Time,
};

VOID
   main()
{
   ULONG Output(), Open(), File = 0;
   struct MsgPort *RendevousPort = NULL;
   register int messages_pending = 0;
   VOID Report_Stats();

   if ( !(RendevousPort = CreatePort(0L, 0L)) )
   {
      Puts("Can't create message port!");
      exit(1);
   }

   /* Note that this file gets closed by the Child, even if the Child
    * fails to run (i.e., ASyncRun() returns an error code).
    */
   if ( !(File = Open("TEST_FILE", MODE_NEWFILE)) )
   {
      Puts("Can't create test file!");
      exit(1);
   }

   /* We must initialize the ReplyPort of the Zombie Message to the port */
   Status_Zombie.zm_ExecMessage.mn_ReplyPort =
      Type_Zombie.zm_ExecMessage.mn_ReplyPort = RendevousPort;

   /* We inform ASyncRun() that we wish a reply by simply giving it a
    * a message to mail us.
    */
   Status_PCB.pcb_LastGasp = &Status_Zombie;
   Type_PCB.pcb_LastGasp = &Type_Zombie;

   /* We use a custom structure to record the name of the process */
   /* And also to record its status, and its departure time */
   StatusInfo.Return = TypeInfo.Return = -1;
   StatusInfo.Name = "Status";
   TypeInfo.Name = "Type";

   Status_Zombie.zm_UserInfo = (ULONG)&StatusInfo;
   Type_Zombie.zm_UserInfo = (ULONG)&TypeInfo;

   /* Finally, we are going to let status use our own output handle */
   Status_PCB.pcb_Output = Output();
   /* Therefore, we do not want a terminating status program to close
   * our output handles so we do:
   */
   Status_PCB.pcb_Control |= PRF_SAVEIO;

   /* And Type will use TestFile, and will close it when done */
   Type_PCB.pcb_Output = File;

   /* the setup is basically done, now just fire off the tasks */
   /* Note that we want to wait ONLY for tasks that ASyncRun() tells
    * us have been succesfully spawned.  If there is a problem, ASyncRun
```

```
 * returns a negative error code.
 */

DateStamp( &TypeInfo.StartTime );     /* Get launch time */

if ( (TypeInfo.Return = ASyncRun("Type", "Sync.c Opt N", &Type_PCB)) >= 0)
{
   messages_pending++;

   DateStamp( &StatusInfo.StartTime );
   if ( (StatusInfo.Return = ASyncRun("Status", "Full", &Status_PCB)) >= 0)
      messages_pending++;

   while (messages_pending)
   {
      /* Do other processing, allowing children to run in background */
      WaitPort(RendevousPort);       /* wait for a child to halt */
      GetMsg(RendevousPort);         /* pull the message */
      messages_pending--;
   }
}

/* Note that it would be a mistake to do a Close( File ) here, since
 * File will be closed whether or not Type ran.
 */

/* now report statistics */
Report_Stats( &Status_Zombie );
Report_Stats( &Type_Zombie );

DeletePort(RendevousPort);
exit(0);
}

/* report termination statistics for processes */

VOID Report_Stats( z )
   struct ZombieMsg *z;
{
   register struct MyUserInfo *u;

   u = (struct MyUserInfo *)z->zm_UserInfo;

   if ( u->Return < 0)
   {
      Printf("Process %s was not run: Error = %ld\n", u->Name, u->Return);
      return;
   }
   Printf("CLI %ld named %s ", z->zm_TaskNum, u->Name);

   /* convert datestamp to human readable form */

   dt.dat_Stamp = u->StartTime;

   if ( !StamptoStr( &dt ) )
      Printf("Launched at %s on %s\n", dt.dat_StrTime, dt.dat_StrDate);
   else
      Puts("<oops> <oops> <oops>");

   dt.dat_Stamp = z->zm_ExitTime;

   if ( !StamptoStr( &dt ) )
      Printf("Exited at %s on %s\n", dt.dat_StrTime, dt.dat_StrDate);
   else
      Puts("<oops> <oops> <oops>");

   /* IF an error, report exit status and IoErr() */

   if (z->zm_ReturnCode != 0)
   {
      Printf("Process exited with status = %ld, Result2 = %ld\n",
            z->zm_ReturnCode, z->zm_Result2);
   }
}
```

## Listing 4: wbproc.c - Creating a non-CLI process

```
/* wbproc.c -- program that simulates Workbench.
 *             Demonstrates the PRB_NOCLI bit which is used to
 *             create non-cli processes with ASyncRun().
 *
 *      Copyright (c) 1988 by Scott Ballantyne -=+SDB+=-
 *      Use and abuse at will.
 */

#include <exec/types.h>
#include <libraries/arpbase.h>
#include <arpfunctions.h>
#include <workbench/startup.h>

struct ProcessControlBlock PCB =
{
   10240,
   0,
   PRF_NOCLI,            /* Set: Not a CLI */
};

struct WBStartup Fake; /* Our fake workbench message */

VOID
   main()
{
   struct MsgPort *WBPort;

   if ( !(WBPort = Fake.sm_Message.mn_ReplyPort = CreatePort( 0L, 0L )) )
   {
      Puts("Couldn't obtain message port!");
      exit(1);
   }

   /* Tell ASyncRun() we already have the code to run */

   if ( !(PCB.pcb_LoadedCode = Fake.sm_Segment = LoadPrg("Clock")) )
   {
      Puts("Couldn't load program!");
      DeletePort( WBPort );
      exit(1);
   }

   /* Launch the program */

   if ( ASyncRun("Clock", NOCMD, &PCB) >= 0 )
   {
      /* Set the message port or 'process ID' of Child in message */
      Fake.sm_Process = PCB.pcb_WBProcess;

      /* Rest of message can be null, for this program */
      Fake.sm_NumArgs = 0;
      Fake.sm_ToolWindow = NULL;
      Fake.sm_ArgList = NULL;

      /* Now send the message, this will start the program up */

      PutMsg( Fake.sm_Process, &Fake );

      /* At this point, we would normally do other processing - however
         if there were nothing left to do, we would just... */

      WaitPort( WBPort );
      GetMsg( WBPort ); /* Pull the message */
   }
   else /* Failure */
      Puts("Could not launch program!");

   /* Cleanup: Ordinarily, all we would need to do is the DeletePort(),
      but since we used the pcb_LoadedCode variable, we must also
      unload the program. */

   UnLoadPrg( PCB.pcb_LoadedCode );
   DeletePort( WBPort );
}
```

## Listing 5: magic.c - Use part of parent code as sub-process

```
/* Magic.c -- make a piece of me a subprocess, using ASyncRun().
 *
 *      Copyright (c) 1988 by Scott Ballantyne
 *      Use and abuse as you please.
 *
 * Note to Lattice users: You must compile with -y and -v!
 *
 * lc -y -v Magic.c
 */

#include <exec/types.h>
#include <libraries/arpbase.h>
#include <libraries/dos.h>
#include <arpfunctions.h>

typedef void PROCESS;

PROCESS MyProcess();

struct ProcessControlBlock PCB =
{
   6000,                                /* stacksize */
   0,                                   /* pri */
   PRF_CODE | PRF_NOCLI | PRF_SAVEIO,   /* control flags */
};

struct ZombieMsg  ChildMsg;     /* We have to get this before exiting */

VOID
   main()
{
   register struct MsgPort *RendevousPort = NULL;
   register LONG rc;
   register int i;

   if ( !(RendevousPort = ChildMsg.zm_ExecMessage.mn_ReplyPort =
         CreatePort( 0L, 0L )) )
   {
      Puts("Can't create port!");
      exit(1);
   }

   /* Ask to be informed on exit */
   PCB.pcb_LastGasp = &ChildMsg;
   /* Aim ASyncRun() at our code */
   PCB.pcb_LoadedCode = (CPTR)MyProcess;

   /* Spawn the sub process */
   if ( (rc = ASyncRun("MyProcess", NOCMD, &PCB)) >= 0)
   {
      /* You can use the child's MsgPort or not as you prefer */
      /* If you do use it, be careful when the Child is calling */
      /* AmigaDOS processes - they use it too! */
      /* Note that we get this because we also set PRB_NOCLI */
      /* PRB_CODE will not automatically supply it */

      Printf("PARENT: Child's msg port is at %08lx\n", PCB.pcb_WBProcess);

      /* We just count a little, the child and the parent count at the */
      /* same time */
      for (i = 0; i < 300; i++)
         Printf("%d ", i);
      Printf("PARENT: Finished counting\n");

      WaitPort( RendevousPort );
      GetMsg( RendevousPort );
   }
   else
      Printf("PARENT: Could not spawn child, ERROR = %ld\n", rc);
   Printf("PARENT: Child exited with status = %ld\n", ChildMsg.zm_ReturnCode);
   DeletePort(RendevousPort);
}

/* Here is the child process code.  You could do graphics, asyncfile io, */
/* whatever you want.  Here, we just open a file, print a few messages   */
/* to prove that we can do DOS stuff here, and scram.                    */
/* Note that we can't call any compiler library stuff here, we aren't    */
/* entering MyProcess through the normal C startup code, so we aren't    */
/* setup for it, hence the uppercase E in exit().                        */

PROCESS
   MyProcess()
{
   ULONG Open(), MyFile;
   register int i;

#ifdef AZTEC_C
   geta4();
#endif

   if ( !(MyFile = Open("CON:0/0/640/100/Child's IO Window", MODE_NEWFILE)))
     Exit(20L);

   for ( i = 0; i < 100; i++)
     FPrintf(MyFile, "%d ", i);

   FPrintf(MyFile,
         "\nCHILD: I have finished counting, and am about to exit\n");
   /* Delay a little, so they have time to read the message */
   Delay(60L);
   Close(MyFile);
   Exit(0L);
}
```

## Listing 6: dodate.c - Date conversion

```
/* DoDate.c -- Example of how to do canonical Date conversions using
 *             StamptoStr() and StrtoStamp(), and the environment
 *             variable 'dateformat'.
 *
 *      Copyright (c) 1988 by Scott Ballantyne
 *      Use and abuse as you please.
 */
#include <exec/types.h>
#include <libraries/arpbase.h>
#include <arpfunctions.h>

/* Storage for strings */

BYTE EnvBuf[ LEN_DATSTRING ]; /* More than we need for this one... */
BYTE DayStr[ LEN_DATSTRING ];
BYTE DatStr[ LEN_DATSTRING ];
BYTE TimStr[ LEN_DATSTRING ];

/* Both StamptoStr() and StrtoStamp() use the following structure */
/* to control input and output formats, and return conversions to you */

struct DateTime MyDT =
{
   0, 0, 0,             /* A DOS datestamp structure */
   0, 0,                /* Format and flags, set later */
   &DayStr[0],          /* Ascii storage */
   &DatStr[0],
   &TimStr[0],
};

/* Just to drive the point home, some environment dependent prompts */
BYTE *Prompts[] =
{
   "DD-MMM-YY, i.e. 23-Jan-88",
   "YY-MM-DD, i.e. 88-01-23",
   "MM-DD-YY, i.e. 01-23-88",
```

```
   "DD-MM-YY, i.e. 23-01-88",
};

/* Finally, a buffer for ReadLine() */
BYTE Buffer[ MaxInputBuf ];

VOID
   main()
{
   register char *cp = EnvBuf;

   /* First determine the users preferences */
   if ( Getenv( "dateformat", EnvBuf, LEN_DATSTRING ) )
   {
      /* Ok, user has expressed a preference, so we must figure out */
      /* what they want.  First, check for a leading '-' */
      if ( *cp == '-' )
      {
         /* Obviously a hardcore computer user, they don't want */
         /* Ascii substitutions for dates - so disable it */
         MyDT.dat_Flags |= ( ~DTF_SUBST );

         /* Advance past '-' for Atol() */
         cp++;
      }
      else
         MyDT.dat_Flags |= DTF_SUBST;

      /* And figure out the format - Atol() returns 0 on bad digits,
         and the conversion routines ignore anything greater than
         FORMAT_MAX, defaulting to FORMAT_DOS, so we are in peaches
         and cream */
      MyDT.dat_Format = Atol( cp );
   }
   else
      MyDT.dat_Format = FORMAT_DOS;       /* Do the default */

   /* Note that the user can enter things like Monday here - typically,
      in a date reading/setting function like this one, we will want
      that to refer to 'next monday'. You can do that like so: */
   MyDT.dat_Flags |= DTF_FUTURE;

   do
   {
      Printf("Please enter the date (%s): ",
            Prompts[ (MyDT.dat_Format > FORMAT_MAX) ? 0 : MyDT.dat_Format]);

      ReadLine( Buffer );
      /* Now copy to appropriate position in MyDT */
      strncpy( MyDT.dat_StrDate, Buffer, 10);
      Printf("Please enter the time (HH:MM:SS): ");
      ReadLine( Buffer );
      strncpy( MyDT.dat_StrTime, Buffer, 10);
   }
   while ( StrtoStamp( &MyDT ) != 0 );
   Printf(" Converted your values to %ld Days %ld Minute %ld Ticks\n",
         MyDT.dat_Stamp.ds_Days,
         MyDT.dat_Stamp.ds_Minute,
         MyDT.dat_Stamp.ds_Tick );

   /* For the other function, we will just read the current system time,
      convert it and display it */
   DateStamp( (struct DateStamp *)&MyDT ); /* Get system time */

   if ( StamptoStr( &MyDT ))
      Puts("<oops> <oops> <oops>");
   else
   {
      Printf("Current system time is %s %s\n",
            MyDT.dat_StrDate, MyDT.dat_StrTime );
   }
   /* The end */
}
```

## Listing 7: freq.c - Using the file requester

```
/* freq.c -- ARP demo program for File Requester (FileRequest()).
 *      Illustrates:
 *          Simplest example for using FileRequest()
 *          More advanced use of FileRequest including:
 *              Use in a CUSTOMSCREEN
 *              Using the special FuncFlags values and fr_Function
 *                      to customize the requester for your needs
 *
 *      Copyright (c) 1988 by Charlie Heath for Transactor magazine
 *      Use and abuse as ye please.
 */

#include <exec/types.h>
#include <libraries/arpbase.h>
#include <intuition/intuition.h>

#ifdef AZTEC_C
#include <functions.h>
#include <arpfunctions.h>
#else
#include <proto/arp_pragmas.h>
#include <proto/intuition.h>
#endif

/* The FileRequester data structure, which is passed as the calling argument,
 * needs FILENAME and DIRECTORYNAME character buffers.
 * Note the pre-initialized default values in the FileRequester structure
 */

char    FileName[FCHARS+1];
char    DirName[DSIZE+1];

struct FileRequester MyFileRequest =
        {
        "My Prompt String",     /* Hailing Prompt              */
        FileName,               /* Pointer to file name buffer */
        DirName,                /* Pointer to dir name buffer  */
        0,                      /* Calling Window or NULL      */
        0,0,                    /* Flags for special control   */
        0,                      /* Special function (none here) */
        0                       /* Reserved                    */
        };

IMPORT struct ArpBase   *ArpBase; /* This is included for Lettuce      */
                                  /* it could be put in arp_pragmas.h  */

void main(argc,argv)
int     argc;
char    *argv[];
{
/* The first example is really simple - just use the pre-initialized
 * data structure defined above, and call FileRequester.
 * This is what you'd do if you haven't opened an Intuition window,
 * and you just want a vanilla file requester to appear on the
 * Workbench screen
 */

Printf("First open the file requester with no calling window...\n\n");
if ( FileRequest( &MyFileRequest) == 0L )
        {
        Printf("User Cancelled File Requester!!!\n");
        }
else
        {
        Printf("User selected FILENAME: %s DIRECTORY: %s\n",FileName,DirName);
        }


/* The next example is a lot more complex, so it has been put into a
 * function all by itself for clarity
```

```
 */

Printf("Now do a fancier call to the file requester...\n\n");
DoFancyStuff();
}

/*********************************************************************/
/* The rest of the code and data here is an example which uses many of  */
/* the programmer customizations for the file requester.               */
/*********************************************************************/

/* First we'll need a custom screen with a window opened inside it. */

struct TextAttr MyTextAttr = { (UBYTE *)"topaz.font",TOPAZ_EIGHTY,0,0 };

struct NewScreen MyNewScreen =
        {
        0,50,320,200,3,
        1,2,
        0,CUSTOMSCREEN,&MyTextAttr,
        (UBYTE *)"My Custom Screen",
        0,0
        };

struct NewWindow MyNewWindow =
        {
        10,10,300,150,
        1,2,
        REFRESHWINDOW+NEWSIZE+ACTIVEWINDOW,
        ACTIVEWINDOW+SMART_REFRESH+WINDOWDEPTH+WINDOWSIZING+WINDOWDRAG,
        0,0,
        (UBYTE *)"My Window",
        0,                      /* Custom screen later...       */
        0,
        20,20,1000,1000,
        CUSTOMSCREEN
        };

/* Add in this gadget, just for grins   */

#define MYGADGETID      1

struct IntuiText drtb = {1,2,JAM2,88,3,&MyTextAttr,
                        (UBYTE *)"Custom Gadget!!!",NULL};
struct Gadget srg3 = { NULL , 5,128, 310,14,
    GADGHCOMP, RELVERIFY | TOGGLESELECT, BOOLGADGET,
    NULL, NULL,&drtb,NULL,NULL, MYGADGETID, NULL };

/*********************************************************************/
/*                                                                   */
/* MyFunc(Mask,Object)                                               */
/*      This function gets is the fr_Function we'll set up to process  */
/* special events selected by fr_Flags within the FileRequest call.  */
/*                                                                   */
/*********************************************************************/
MyFunc(Mask,Object)
ULONG   Mask;
CPTR    *Object;
{
switch ( Mask )
        {
        case FRF_NewWindFunc:
                /* Note - you will need to do something like this if    */
                /* you want to use FileRequest() in a LOWRES screen     */
                ((struct NewWindow *)Object)->LeftEdge = 0;
                ((struct NewWindow *)Object)->Height = 128 + 15;
                break;

        case FRF_AddGadFunc:
                AddGadget( ((struct Window *)Object),&srg3, 0L);
                break;

        case FRF_GEventFunc:
/* Note - it is not highly advised to do complex library function calls
 * such as Printf within MyFunc, because there might possibly be interactions
 * or program stack overflow issues which might not be obvious.  We have
 * used Printf here for demonstrative purposes, but don't recommend it
 * for daily consumption.
 */
                Printf("Gadget Event Function called\n");
/* Note, sigh, this return code doesn't work in V1.1 arp.library        */
                return(TRUE);

        case FRF_DoMsgFunc:
                Printf("Message Event Function called\n");
                ReplyMsg( Object );
                break;
        }
}

/*********************************************************************/
/*                                                                   */
/* DoFancyStuff()                                                    */
/*      This function does all the fancy things, opening a custom    */
/* screen and window, setting up special flags, and processing them  */
/* with MyFunc above.                                                */
/*                                                                   */
/*********************************************************************/
DoFancyStuff()
{
struct Screen   *MyScreen;
struct Window   *MyWindow;

ULONG   retval;

MyScreen = OpenScreen(&MyNewScreen);
if ( MyScreen == 0L )
        {
        Printf("Can't open custom screen, Oh No!!!!\n");
        return(1);
        }

MyNewWindow.Screen = MyScreen;
MyWindow = OpenWindow(&MyNewWindow);
if ( MyWindow == 0L )
        {
        CloseScreen(MyScreen);
        Printf("Can't open custom window, Oh Doubleno!!!\n");
        return(2);
        }

/* OK, we've got the CUSTOMSCREEN open with a window in it...
 * Now set up the FileRequester structure for the special flags
 * conditions we want to use.  Note that we're leaving most of the
 * default values in the FileRequester structure as they were
 * initialized before...
 */

MyFileRequest.fr_Function = (VOID (*)())MyFunc;
MyFileRequest.fr_Window   = MyWindow;
MyFileRequest.fr_FuncFlags = FRF_AddGadFunc+FRF_GEventFunc+
                        FRF_NewWindFunc+FRF_DoMsgFunc;

retval = (ULONG)FileRequest( &MyFileRequest );

CloseWindow(MyWindow);
CloseScreen(MyScreen);

if ( retval == 0L )
        {
        Printf("User cancelled file request in custom screen!!!\n");
        }
else
        {
        Printf("(In CUSTOMSCREEN...)\n");
        Printf("User selected FILENAME: %s DIRECTORY:%s\n",FileName,DirName);
        }
```

# The ANSI Standard

## *A summary for the C programmer*

**Eric Giguere**
*Copyright © 1987 Eric Giguere*

*Eric Giguere has been involved with the design of an ANSI-conformant C compiler at the Computer Systems Group at the University of Waterloo, where he is a third-year computer science student in the co-op program.*

Programming languages are constantly evolving and diversifying. The C language is no exception, especially due to its increased popularity in recent years. The original specification document for C, *The C Programming Language* by Brian Kernighan and Dennis Ritchie (commonly referred to as K&R), is now almost ten years old. K&R has served as the C programmer's 'bible', the de facto standard for C. But as the language evolved, the need for a formal language standard became apparent.

Under the auspices of the International Standards Organization (ISO), the American National Standards Institute (ANSI) began the preparation of such a standard under the leadership of the X3J11 Technical Committee. The proposed standard is now in final draft form and is expected to be approved by ANSI and ISO this year.

This document is a summary of the proposed standard's major changes to the language as it pertains to the C programmer. All information is drawn from official X3J11 documents: the *Draft Proposed American National Standard for Information Systems - Programming Language C*, and the accompanying *Rationale for Draft Proposed National Standard for Information Systems - Programming Language C*. (These publications will be referred to in this article as the *Standard* and the *Rationale*, respectively.)

This article is not a criticism of the *Standard*, only a commentary. Readers should also be aware that changes may occur to the *Standard* before its final acceptance by ANSI.

Few programmers have either the interest or the time to wade through the actual text of the draft *Standard* in its entirety. What interests them are the changes that the *Standard* makes to what was defined in K&R and how it affects their current programs. By summarizing these changes, this article provides a quick reference that the average C programmer can read and understand in one session.

### What to expect from the *Standard*

The *Standard* is not creating a new language definition. Its purpose, to quote the *Rationale*, is to "codify common existing practice". This means that the fundamental structure and syntax of the language as described in K&R has been left unchanged. The *Standard* has instead tried to unify the diverse extensions and dialects (the "existing practice") that have grown over the years into a single cohesive language definition. Existing practice is often inconsistent, however, so many compromises have had to be made.

Perhaps the most important thing to remember about the *Standard* is that it is not intended to invalidate existing C code. Existing programs should still compile under an ANSI-conformant compiler with only minor changes.

### Reserved identifiers

The following keywords have been added to the language:

*const, enum, signed, void, volatile*

Explanations for each follow below. In addition, the *identifier* entry has been deleted from the list of reserved identifiers as it was never implemented by K&R or subsequent versions of C.

### Data types

Major language changes occur with respect to data types.

**Integers:** The list of available integer types has been widened to include *signed char* and the following variations:

unsigned int
signed int
unsigned long
signed long

```
unsigned short
signed short
```

The types *long int* and *short int* may also be used as variations of *long* and *short*, respectively. As would be expected, the declarations:

```
signed x; unsigned y;
```

may be used as shorthand for *signed int* and *unsigned int*. Note that at least one storage class (*auto*, *register*, *static*, *extern*) or type specifier must be present when declaring a variable. A declaration of the form *x;* is no longer allowed and must be replaced with *int x;* to compile.

Whether or not a simple *char* is considered to be signed or unsigned is left up to the compiler.

**Floats:** The new type *long double* has been added for more precision. But like any long type, it is only guaranteed to be at least as large as the next smaller type, in this case a *double*.

The type *long float* (a synonym for *double*) is now invalid.

**Structures and unions:** Member name spaces are now unique. That is, two different structures or unions may contain members of the same name without fear of conflict.

Structures and unions may now:

- be assigned to another of the same type
- be initialized when declared with the auto storage class
- be passed as function parameters and return values

**Enumerations:** Already available in most compilers, enumerations have been officially added to the language. An enumeration is a way of declaring a set of integer constants. The declaration:

```
enum colours { RED, BLUE, GREEN };
```

would declare *colours* as an enumeration tag representing the integer constants RED, BLUE, and GREEN. These enumeration constants are given integer values starting at 0 and increasing by 1 with each identifier. An enumeration constant has type *int* and may be used whenever an integer may be used. In use it is equivalent to the following:

```
#define RED 0
#define BLUE 1
#define GREEN 2
```

Variables may be declared to have enumeration type:

```
enum colours x, y;
```

declares *x* and *y* to be integer variables capable of holding an enumeration constant of type *colours*. In practice, no checking is done to make sure only enumeration constants are used, so the following assignments are equivalent:

```
x = BLUE;
x = 1;          /* defeats purpose */
```

(Enumeration constants are not restricted to upper case, but upper case is a recognized convention for constants.) Constant values may be directly assigned within an enumeration as well:

```
enum relation {
    EQUAL = 1, LESS_THAN = 2, GREATER_THAN = 4 };
```

If no value is specified for a given identifier, the constant is taken to have the value of the last constant plus one.

**Void type:** The *void* data type has been added to indicate that an expression returns no value. No variables can be declared with such a type, but expressions may be cast to *void*. For example, the following declaration:

```
(void) printf( "hello world\n" );
```

specifically indicates to the compiler that the return value from *printf* (an integer) is to be ignored. As such, the following statement is illegal:

```
a = (void) func();  /* illegal! */
```

since the assignment operator expects a value to be returned for assignment.

*Void* pointers and *void* functions are discussed below, and later on under 'Function Declarations'.

**Pointers and arrays:** Pointers are no longer synonymous with *int*. Pointers may only be compared with or assigned:

- the integer value 0 (used to define a null pointer)
- a pointer of the same type
- a pointer of generic type (a *void* pointer)

A *void* pointer is a pointer that has no base type - that is, it points to a type of unknown size - and is declared using the syntax:

```
void *ptr;
```

Indirection through a *void* pointer is not allowed; it must be cast to an appropriate pointer type first.

Arrays with storage class *auto* may now be initialized. If specified, the size of an array must be an integral expression greater than zero.

**Special modifiers:** The *Standard* makes available the two attributes *const* and *volatile* for use as type modifiers.

An object declared to be *const* cannot be modified (assigned to, incremented, decremented) by a program. Thus the following code is invalid:

```
const int x;
/* ... */
x = 2;          /* ILLEGAL!!! */
```

Initialization, however, is allowed:

```
const unsigned int masks[] = {
    0x0000, 0x00ff, 0xff00, 0xffff };
```

A *const* object (if it is *static*) may be data that are put into read-only memory. Declaring such data as *const* allows the compiler to diagnose any (futile) attempts at modifying them.

A *volatile* type is one that may be modified outside of program control. Memory-mapped I/O ports are a typical example. Declaring an object as *volatile* indicates that the compiler should always fetch the object's value from its actual memory location - it may have changed since the last access. (This prevents optimizations which could store the value in a register and possibly return erroneous results.)

```
volatile char *port1 = 0x00f3; /* ptr to I/O port */

while( *port1 & DATA_FLAG ) /* needs to be volatile */
    clear_io();
```

The *const* and *volatile* modifiers may be used (singly or together) in combination with any other valid type specifiers.

Pointers may also be declared to be *const* or *volatile* through the use of special syntax best demonstrated by example:

```
const int *x;
int *const y;
```

Here, *x* is declared to be a pointer to a constant integer, whereas *y* is declared to be a constant pointer to an integer. The distinction lies in the placement of the *const* attribute. Consider the following statements:

```
x = NULL;   /* okay */
*x = 0;     /* error */
y = NULL;   /* error */
*y = 0;     /* okay */
```

Because *x* is a pointer to a *const int*, the value it points to may not be changed. Similarly, because *y* is a *const* pointer to an *int* it may not be modified, though the value it points to may. Similar rules exist for *volatile*, which may also be used in conjunction with *const*.

**Bit-fields:** Bit-fields may now be of type *int*, *unsigned int* or *signed int*. Whether or not the high-order bit of an *int* bit-field is to be considered a sign bit is implementation-defined.

**Vacuous definitions:** A vacuous definition consisting only of a *struct* or *union* specifier with a tag name is now allowed. Its purpose is to hide any outer declaration of the same name in the current block:

```
struct a
    {
        int x;
    };

int func()
    {
        struct a st1;   /* struct defined above */
        struct a;               /* vacuous definition */

        struct b
            {
                struct a *y;    /* refers to next struct */
            } st2;
        struct a
            {
                struct b *z;
            } st3;

        st1.x = 1;
        st2.y = &st3;  /* &st1 would give warning */
        st3.z = &st2;
    }

/* old struct is back in scope */
```

Here the member *y* of *st2* is a pointer to the second *struct a*, which is defined below it. If the initial *struct a;* had not been present, *y* would instead have been a pointer to the *struct a* defined in the previous (enclosing) scope level.

**Conversions and promotions:** The integral promotions are defined thus: the *char*, *short* or bit-field types, with or without the *signed* or *unsigned* modifiers may be used wherever an *int* is expected. The values will be converted to *int* if possible; otherwise they will be converted to *unsigned int*.

The usual arithmetic conversions used with most binary operators have been modified to reflect the new types available to the programmer. The conversions are applied to the operands of the expression in the following order:

- First, if one is *long double*, the other is converted to *long double*.
- Otherwise, if one is *double*, the other is converted to *double*.
- Otherwise, if one is *float*, the other is converted to *float*.
- Otherwise, the integral promotions are performed. Then if one is *unsigned long int*, the other is converted to *unsigned long int*.

- Otherwise, if one is *long int* and the other is *unsigned int*, if a *long int* can represent all the values of an *unsigned int*, the *unsigned int* is converted to *long int*, otherwise both are converted to *unsigned long int*.

- Otherwise, if one is *long int*, the other is converted to *long int*.

- Otherwise, if one is *unsigned int*, the other is converted to *unsigned int*.

- Otherwise, both operands must have type *int*, as required by the integral promotions.

The *Standard* also specifies other rules regarding the conversion of *unsigned* to *signed*, integer to floating and floating to floating. These rules for the most part are unchanged from the K&R definition or are (supposedly, at least) common sense in nature.

**Minimum type limits:** Any compiler conforming to the *Standard* must also respect the following limits with respect to the range of values any particular type may accept. Note that these are lower limits: an implementation is free to exceed any or all. Note also that the minimum range for a *char* is dependent on whether or not a *char* is considered to be *signed* or *unsigned*.

- number of bits in char: 8
- range for signed char: -127 to +127
- range for unsigned char: 0 to 255
- range for short int: -32767 to + 32767
- range for unsigned short int: 0 to 65535
- range for int: -32767 to +32767
- range for unsigned int: 0 to 65535
- range for long int: -2147483647 to +2147483647
- range for unsigned long int: 0 to 4294967295
- precision for float: 6 digits
- precision for double: 10 digits
- precision for long double: 10 digits

The *Standard* also specifies that these limits should be present as preprocessor macros in the header file <limits.h>.

**Data objects**

Changes have occurred in this area mainly with respect to variable (object) linkage and initialization.

**Initialization:** Objects declared as either *static* or *auto* may be initialized by following the declaration with an '=' and an initialization expression. External (inter-module) objects are discussed below under 'Linkage'.

If no initialization is given for a *static* object, all arithmetic types in the object are assigned 0 and all the pointers are set to NULL. If no initialization is given to an *auto* object, the initial value is undefined.

All initializers for either *static* objects or arrays, unions and structures must be constant expressions.

Unions may be initialized; the initialization value is assigned to the first member of the union.

The initialization expression for a scalar (integral, floating-point or pointer) object may optionally be enclosed in braces. Braces must enclose the initialization expressions for arrays, structures and unions. There can be no more initializers in an initialization list than there are objects to be initialized. An array of or pointer to *char* may be initialized with a string constant.

**Linkage:** (Note: in this section, "object" refers to an object declared outside of any function.)

The linkage of an object determines its scope within the program. An object with external linkage is known to all files in a program. An object with internal linkage is known only to the file in which it is defined. Current C compilers often differentiate between the two in incompatible ways, an issue that the *Standard* resolves.

An object is said to be defined if it includes an initializer. A defined object has internal linkage if the storage class *static* is specified; otherwise it has external linkage. An object can only be defined once.

If no initializer is present, any object declaration without the *extern* modifier constitutes what is known as a tentative definition of the object. If an actual definition for the object is encountered in the same file, all tentative definitions are considered to be simple declarations referring to that object. Otherwise the first tentative definition is considered to be an actual definition with initializer equal to 0.

The simplest way to declare an externally-linked object is to define it in one file (with or without an initializer), and reference it in all others through the use of an appropriate *extern* declaration.

**Function declarations**

Perhaps the most useful additions to the language definition occur in the area of function declarations, function definitions and variable parameter lists.

**Function definitions:** The *Standard* now allows the types of formal parameters to be specified within the actual function declaration at the start of a function definition. This new-style definition form resembles more closely languages such as Pascal and Modula-2:

```
main( int argc, char *argv[] )
    {
        /* .... */
    }
```

If this style is used, a type must be specified separately for each formal parameter in the argument list. Mixing the new-style with the K&R-style is not allowed.

This style is intended by ANSI to become the favoured style; a future *Standard* may disallow the K&R-style definitions.

Functions may also be defined as explicitly having no return values. Such functions are called *void* functions and are defined using the type *void*:

```
void func( int a )
   {
      /* ... */
   }
```

Such functions may not return an expression. If no type is explicitly specified, the function still defaults to *int* to retain compatibility with K&R.

**Function declarations and prototypes:** To go along with the new-style function definitions, function type declarations may also include a list of formal parameters. These parameters consist of type declarations with or without identifiers. Identifiers are cosmetic only, and go out of scope at the end of the declaration. Examples:

```
int main( int, char *[] );
extern char *strcpy( char *dest, const char *src );
```

(You must be consistent in your declarations. Each declaration of a function should agree with all previous declarations in the number and types of parameters.) The following declarations illustrate two special cases:

```
extern int getchar( void );
extern int printf();
```

The first case explicitly declares that the function getchar() does not take any parameters - that the parameter list is empty or void. The second declaration states that no information is known on the number and types of any formal parameters. This, of course, is to provide compatibility with K&R.

A function declaration that does provide the number and types of parameters is called a *function prototype*. The addition of prototypes to C allows for stricter type-checking by the compiler. When a prototype for a function has been declared, each subsequent call to that function is checked to make sure that the correct number of arguments has been supplied. As well, the type of each argument is compared against what was declared in the prototype. If different, the argument is converted to the required type as if it had been assigned to an object of that type. The default argument promotions (*char* to *int*, *float* to *double*, etc.) aren't performed when a prototype is in scope.

If a function prototype occurs in the same file as the definition for that function, both the prototype and the definition must agree exactly if the definition is new-style. In K&R-style definitions, the formal parameters are first widened by the default argument promotions and then compared to the prototype(s).

If no prototype occurs in the file, the function definition itself serves as a prototype for the code following it.

**Variable parameter lists:** Certain C functions are designed to take a variable number of parameters. Unfortunately, different compilers use different schemes for handling such situations, and what works in one implementation may not work elsewhere. The *Standard* therefore provides for the explicit declaration of such functions, and portable facilities for handling them. A function taking a variable number of parameters is defined by ending the parameter list (new-style only) with an ellipsis:

```
int printf( const char *format, ... )
   {
      /* .... */
   }
```

Thus the only thing known about printf() is that it takes at least one parameter, the type of which is a pointer to *const char*. Prototypes may also be declared in this fashion:

```
extern int sprintf( char *dest, const char *format, ... );
```

The compiler will then make sure that each call to sprintf() has at least two arguments, both of which are pointers to *char*.

The arguments themselves are accessed through the use of special macro facilities defined in the header file <stdarg.h>, which is not described in this article.

**The preprocessor**

The C preprocessor, long recognized as an integral part of the language, has benefited from a number of additions and clarifications.

**New directives:** The *#elif* directive has been added as a shorthand form of the *#else #if* sequence.

The special identifier *defined* is reserved during an *#if* or *#elif* so that:

```
#if defined( NULL )
#if !defined( TRUE )
```

are equivalent to:

```
#ifdef NULL
#ifndef TRUE
```

Also new on the list are the directives *#error* and *#pragma*. The first of these produces an error message at compile-time; the second is implementation-defined in its use and effects.

**File inclusion:** Besides the two allowable forms:

```
#include <fname1>
#include "fname2"
```

a third form:

```
#include fname3
```

is acceptable provided *fname3* is a macro that expands into one of the other two forms.

**Predefined macros:** Five macros are predefined in the *Standard*, all of which are expanded to their appropriate values upon file compilation.

| | |
|---|---|
| __LINE__ | is the current line number in the source file. |
| __FILE__ | is the current filename. |
| __TIME__ | is the current time of compilation. |
| __DATE__ | is the date of compilation. |
| __STDC__ | is the decimal constant 1. Its definition indicates an ANSI-conformant compiler. |

**Macro operators:** Two new operators have been added for use within a macro replacement string.

The ## operator concatenates two adjacent preprocessor tokens (a preprocessor token is any consecutive series of non-blank characters). The # operator places the parameter following it in string form. For example, consider these definitions:

```
#define str(s)    # s
#define cat(a,b)  a ## b
```

The following macro calls:

```
str( printf( "hello" ) )
cat( x, 2 )
```

result in:

```
"printf( \"hello\" )"
x2
```

(Notice how any quotes within the parameter passed to *str* have been altered so as to not affect the new string.)

**The C library**

A standardized library of routines is a valuable resource for programmers and also enhances portability. The *Standard* defines such a library, which is too large to describe here in detail. The Standard Library is based on the library compiled by /usr/group, a UNIX® users' group, with all the UNIX dependencies thrown out.

The Standard Library also provides a set of standard library headers. These headers provide function prototypes for the set of routines that make up the library and define commonly-used macros. As well, the functions themselves have been changed so as to be invariant to the default promotions; that is, the parameters passed to the function will always be of the same type, whether a prototype is in scope or not.

Macros may also be defined in a header file to take the place of actual calls to library routines. However, the library routines themselves must exist, as the macros may be subjected to an *#undef* directive by the user at any time.

Among the most notable additions to the library are variable argument handling, numeric limits information, and locale (the current environment) information.

K&R library functions have also been converted to the new style and syntax, so that *malloc*, for example, now returns a *void* * rather than a *char* *.

**Miscellaneous**

Numerous other changes and additions have occurred throughout the language. Here is a brief summary of the more important points:

- the escape characters '\a' and '\v' have been added for alarm (bell) and vertical tab
- a series of special trigraph characters has been added as equivalents to ASCII characters that may not appear in the character sets of other countries; a trigraph is a three-character sequence starting with *??*
- the suffixes *u* and *l* may be used with integer constants to specify unsigned and long values; both may be used together to specify unsigned long
- the suffixes *f* and *l* may be used with floating-point constants to specify float and long double values
- if the high bit of an octal or hex constant is set, it is considered to be unsigned
- adjacent strings separated only by white space are concatenated
- the unary plus ('+') has been added to force the evaluation of an arithmetic expression to occur before any others
- a function can be invoked through a pointer using either the K&R-style syntax *(*fp)()* or the new-style *fp()*
- external identifier length significance is still 6 characters with no case sensitivity, for compatibility with existing linkers
- internal identifier length significance is a minimum of 31 characters (case sensitive)

# Internationalizing Your Programs

## *The art of well rounded applications*

**by John Toebes**

*John Toebes is a senior systems analyst at SAS Institute Inc. in Cary, North Carolina, and the author of the Lattice C V4.0 compiler. He has been programming the Amiga since 1985, and is the coordinator for The Software Distillery, which is responsible for many popular freely redistributable programs including Hack, PopCLI, MemWatch and Blink. He can be reached on Bix (JTOEBES), and the Software Distillery BBS at (919) 471-6436.*

The power and flexibility of the Amiga has made it an international machine with a wide variety of configurations. Even with such a diverse audience, there are some surprisingly simple tricks that make it easier to cope with these individual environments.

There are three basic areas often overlooked by programmers that require special consideration when programming for international environments:

1) Screen configurations
2) Keyboard configurations
3) National languages

With just a small amount of work, the first two of these are really problems that shouldn't exist. The last one is an area where you can let your creativity shine. Let's take a tour of what can be done.

### Day 1 - Seeing the sights

The Amiga is able to succeed as a desktop video machine because of its support of video standards. With this come two different types of video displays - the NTSC for the United States, and PAL for European countries. On an NTSC Amiga, the standard screen resolution is 200 lines of 320 or 640 pixels, while a PAL Amiga provides 256 lines of 320 or 640 pixels. A program limited to NTSC proportions will only cover 80% of the screen when run on a PAL Amiga - not a pretty sight. Conversely, running a PAL-specific program on an NTSC machine obscures 20%.

Video differences cannot be dismissed by simply saying a program will only run under one or the other video standards, because there are factors that can cause differences on a single video standard. For example, *MoreRows* is a program that allows the general user to increase the resolution of the Workbench screen; I am writing this article on a Workbench screen with a resolution of 670 by 460 (unfortunately my favourite editor - mg1b - doesn't recognize the extra lines/columns this offers). Also, one must consider interlace, which effectively doubles the screen height. Lastly, there are new monitors coming out (the 2024, for example) which are capable of displaying much higher resolutions. You certainly don't want a program limited to displaying only on a small portion of your new 1024 by 800 pixel monitor.

To deal with all these options, there are a few simple rules to follow:

1) *Never* hard-code a screen or window size or limit. A good example of what *not* to do can be found in the second example in "Intuition: The AMIGA User Interface":

```
struct NewScreen NewScreen = {
    0,          /* The LeftEdge should be equal to zero */
    0,          /* TopEdge */
    320,        /* Width (low-resolution) */
    200,        /* Height (non-interlace) */
....
NewWindow.MaxWidth = 640;
NewWindow.MaxHeight = 200;
```

Instead, you should use the values provided in GfxBase to calculate the correct number of rows and columns.

```
/* Calculate size for window taking into account */
/* interlace and any morerows specifications */
rows = GfxBase->NormalDisplayRows;
cols = GfxBase->NormalDisplayColumns;
GetPrefs(&prefs, sizeof(struct Preferences));
if (prefs.LaceWB)
    rows += rows;
```

Although I recommend against it, you can test for NTSC/PAL directly by looking at GfxBase->DisplayFlag and testing for NTSC, PAL or the GENLOCK bits. Relying on this as an indicator of size doesn't handle *MoreRows* or any new video monitor sizes.

This brings up what I consider to be one of the best ways to learn how the Amiga works. Take the time to write simple test programs to explore system data structures and try out simple system calls.

2) Design screen layouts *algorithmically*. Don't simply assume that a gadget goes at a given offset to place it on the bottom. Use GRELWIDTH, GRELBOTTOM, GRELRIGHT and GRELHEIGHT to let the system figure it out for you. We will examine more of this later on.

3) Be prepared for very large screens. As screens get larger, memory becomes tighter. While not inherently display related, this is often the cause of programs which barely fit on an NTSC machine not fitting on a PAL machine.

One of the biggest complaints about having to deal with varying video displays (PAL in particular) is that there is no way to test the different configurations without having both a PAL and an NTSC Amiga. Actually, one can test PAL conformance on an NTSC machine quite readily. Instead of opening up a 640/200 size screen, open up a 640/256 *interlace* screen. Although the screen won't cover the entire display, and there might be some flicker, this will give you the chance to see how the screen layout looks in that situation. Testing NTSC conformance on a PAL machine doesn't even require interlace; just open a screen 200 pixels high.

### Day 2 - Understanding the language

Unlike displays, keyboard differences are harder for a program to be aware of. Each country has its own keyboard and mapping, with keys moved around as well as different character codes appearing. Even in the United States there are three different keyboard mappings to consider, including the Dvorak keymap supplied as USA2.

The Amiga allows you to get input in four ways:

1) Reading from a CON: or a RAW: window. (Note that stdin in a C program typically points to here)

2) Reading from a "console.device" attached to a window

3) Intuition VANILLAKEYS

4) Intuition RAWKEYS.

Of these, only RAWKEYS has any potential for problems; all the others automatically translate the incoming keystrokes appropriately. In the last case, you need to translate the keys according to the current default keymap. To do so, V1.2 has supplied a new routine, RawKeyConvert, which will perform the conversion, including handling of "dead keys" (which provide accents). It is described in detail in the Amiga Enhancer Software manual that comes with the upgrade.

To use it, you need to take the message from RAWKEY and convert it to an input event. You pass this and a buffer to hold the conversion to the RawKeyConvert routine, along with a pointer to the keymap to use. If you pass NULL for the KeyMap, the system will default one for you. Note that you must open the console.device before using this routine. A typical program would need code like:

```
struct Device *ConsoleDevice;
struct IOStdReq devreq;
char keybuf[10];

OpenDevice("console.device", -1, &devreq, 0);
ConsoleDevice = devreq.io_Device;
...
for(;;)
{
/* Wait for the next event to happen */
while ((message = (struct IntuiMessage *)GetMsg(window->UserPort)) == NULL)
   Wait(1 << window->UserPort->mp_SigBit);

   switch(message->Class)
   {
   ...
      case RAWKEY:
         len = KeyConvert(message, keybuf, sizeof(keybuf));
         /* now the converted keystrokes are in keybuf for length len */
   ...

/*********************************************************************/
/* Convert a RAWKEY message to input text using the default keymap for */
/* the console                                                       */
/* Parameters:  msg  - RAWKEY message from Intuition. It must not    */
/*                     have been replied to yet.                     */
/*              buf, len - buffer (with length) to hold converted keys */
/*********************************************************************/
int KeyConvert(msg, buf, len)
strict IntuiMessage *msg;
char *buf;
int len;
{
   static struct InputEvent ievent = {
      NULL, IECLASS_RAWKEY, 0, 0, 0
   };

   if (msg->Class != RAWKEY)
      return(0);

   /* Convert the intuition rawkey into an input event */
   ievent.ie_Code = msg->Code;
   ievent.ie_Qualifier = msg->Qualifier;
   ievent.ie_position.ie_addr = *(APTR *)msg->IAddress;

   return (RawKeyConvert(&ievent, buf, len, NULL);
}
```

Once you have the input characters, you need to ensure that you don't make the mistake of assuming that all characters are less than 127. Some programs that come from UNIX-based machines will mask out the high order bit with a statement of the form:

```
c &= 127;
```

On the Amiga this is a disaster because all 256 characters are used. Many of the international characters have this high order bit set, as do some important symbols like copyright and registered trademark. In general, you should take what the user types in verbatim. Attempting to pretty it up by uppercasing or capitalizing often leads to disaster.

**Day 3 - Speaking the language**

The last area to consider for a well rounded Amiga program is that of handling the various languages available. Unfortunately for this one, there is no cookbook answer to make life simple. The best approach to the problem is to be flexible. Not only doesn't everyone in the world speak English, many people simply don't like the particular way things are worded, or just want to personalize the program to work for them.

You have at least three ways to approach a situation:

1) Do everything in English and expect anyone else who is going to use the program in a different language to take a filezap utility and patch all the messages. This would be fine if all languages were the same length or shorter than English. However, languages such as German often are longer, making this solution impractical at best. This also doesn't answer the issue of screen layout based on length of messages.

2) Don't put out any text messages at all. This is the opposite extreme and provides an excellent opportunity for complete ambiguity. The notion that 'a picture is worth a thousand words' holds quite true when using a small icon to represent an operation. The problem is the user being able to figure out which those thousand words you meant for him to interpret it as. To illustrate, think about some of the traffic signs you take for granted, but only because you have been trained to recognize them.

This doesn't mean to stay away from icons and pictures, just don't rely upon the representation to convey all of the meaning. Graphics are generally more useful when used to indicate operations to perform instead of describing the results of some operation.

3) Allow the user to configure the messages for your program. The best way is to locate a configuration file of messages (preferably associated with the program or in the S: directory) and read it in at the start of the program. In this way, the output and prompt strings are completely flexible. Even for people who speak the same language, this gives them the chance to truly customize the program to their tastes.

While this third solution might seem to be difficult, it is only so for the first string. Once you have set up a mechanism for storing and accessing the messages to be printed, adding another string is simple. In fact, once you have devised something for one program, it generally will work on most programs. The program chatty.c shown below illustrates one such implementation.

**Day 4 - Getting out to meet the natives**

Just because you have a string to print out doesn't mean that the job is complete. Intuition 1.3 brings with it several new fonts, including the proportional Times font. This has the effect of making life a little more interesting. No longer can you simply lay the screen out in terms of character cells, you must program the screen in terms of attributes of the fonts.

To deal with this, Intuition provides several routines for determining the size of text. The easiest to use is IntuiTextLength. You call it with a pointer to an IntuiText structure (exactly what you would use for creating a gadget or menu item). There is one caveat to be aware of here. The font you are referencing (if any) must be in memory. If it was a disk-based font, you must first call OpenDiskFont to lock that font into memory. A TextAttr structure defines the font that you want; a pointer to such a structure goes in the IntuiText structure, and is passed to OpenDiskFont.

IntuiTextLength returns the width of the text in pixels. In order to determine the height, you must look at the YSize field in the opened font structure. So, to find the area of the screen the string 'Box Me' would take in the sapphire/11 font you would code:

```
struct TextAttr mytextattr;
struct TextFont *font = NULL;
static struct IntuiText itext = {
        0, 1, JAM1, 0, 0, NULL, NULL, NULL
};

DiskfontBase = OpenLibrary("diskfont.library", 0);

mytextattr.ta_Name  = "sapphire.font";
mytextattr.ta_YSize = 11;
mytextattr.ta_Style = 0;
mytextattr.ta_Flags = 0;

font = OpenDiskFont(&mytextattr);

itext.ITextFont = &mytextattr;
itext.IText = "Box Me";

width = IntuiTextLength(&itext);
height = font->tf_YSize;

CloseFont(font);
CloseLibrary(DiskfontBase);
```

Note, of course, that you should check for failure and not assume that everything opened okay. Don't forget the .font extension on the name you pass to OpenDiskFont.

**Day 5 - Putting it all together**

Once you understand all the issues involved, it is time to master the techniques in a program. The sample program Chatty.c simply opens up a window the full size of a Workbench screen and prints out random messages from a predefined list. In addition, it provides a single boolean gadget for fast/slow rate of messages. Although quite simple, it illustrates many of the issues covered above.

1) It handles Interlace, *MoreRows* and NTSC/PAL - in fact it will work with any screen size.

2) The messages it writes are customizable by the end user without having to recompile the program.

3) It multitasks quite nicely - even with a loop of printing information it does not do a busy wait.

4) When deselected, it quietly goes to sleep.

5) Its gadget automatically scales to whatever window size is selected.

6) It understands fonts and how large they are.

As we discussed earlier, the problem of handling screen sizes is readily solved by obtaining the information from GfxBase and Preferences. We use this information to open a window the full size of the Workbench screen. As other monitors or default screen sizes are created, this program will automatically adjust. With a program such as a text editor, this may be desirable.

Handling the text display is probably the most difficult part of the program. The program must be able to write out strings that are customizable by the user without having to recompile the program. To accomplish this, they are stored in an external file. However we must be careful *where* we store them, as it must be a place readily associated with the program. In addition, we must be able to function without these messages.

Three obvious places immediately come to mind for storing the messages:

1) In a file in the same directory as the program.

2) In a file in the S: directory

3) In the .info file associated with the program. Choices 1 and 2 are not very good because they require the user to always remember the association between the files. However with option 3, people are already used to copying the .info file (not to mention that Workbench will do this automatically). Given this strong association, this is the logical choice.

Workbench provides us with an excellent place to store information in the .info file: the tool types array *(for more about this, see Rob Peck's article on icons in* ***TransAmi*** *Volume 1 Issue 2)*. To see this, simply select any tool and bring up the info window for it from Workbench. In it you will see on the bottom a box labeled TOOL TYPES with two arrows for scrolling through the tools as well as ADD and DEL gadgets for creating and removing entries. To add a tool, simply click on ADD and enter in the tool information in the string gadget. Existing strings can be modified by locating the tool with the up/down arrows and then clicking on the string gadget. A tool entry looks just like an assignment statement such as:

NAME=text

With this arrangement, we can simply choose names for the configurable options and access them from the program. For Chatty, the following tools are recognized:

OPENFAIL=message to print when it can't open the window
TITLE=title of the window to be opened
FONT=name of font to use
FONTSIZE=Font size to use (an integer)
FONTSTYLE=integer font style.
(1=UNDERLINED 2=BOLD 4=ITALIC 8=EXTENDED)
FONTFLAGS=integer font flags
(2=REVERSE 32=PROPORTIONAL)
GADGET=title for speed control gadget
STRINGA=Random string
...
STRINGZ=Random string

If a particular option is not specified, Chatty will pick an appropriate default. In fact, if the .info file is not present at all, Chatty will still be able to function. When designing a program that can be configured, it is important that a failsafe such as this always be built in. A user is always free to use Workbench to change the strings as he desires.

From the program side of things, it is quite simple to access the strings contained in the .info file. We do this in three steps:

1) Open up icon.library into IconBase so we can use the Workbench icon routines.

2) Locate the .info file. Because of the presence of both CLI and Workbench, we have two situations we must always consider:

a) Under CLI, the program name is stored in argv[0]. This, combined with the current directory stored in our process structure, indicates where we were started from.

b) From Workbench, we receive a startup message that indicates the name of the program and the directory it started

from. Given the name and a directory, we CurrentDir to that directory and then ask Workbench for the DiskObject associated with the file. This is an in-memory copy of the contents of the .info file. From this we get the tool types array which contains all the strings the user typed in.

3) For each string we are interested in, we call the FindToolType() function to search the tool types array. To handle defaulting when the option is not specified, we create a routine getstr() which takes the tool we want to look up and an appropriate default for the tool. For example, our window title comes from:

```
getstr(toolptr, "TITLE", "No title specified")
```

If getstr() cannot find TITLE in the array, it will give us the default string which we end up using. This type of mechanism guarentees that we always have something to work with.

Messages can be handled even more transparently with a macro such as

```
#define M_CANTOPEN (getstr(toolptr, "OPENFAIL", "Can't Open
        the window"))
```

Which allows us to do a

```
        puts(M_CANTOPEN);
```

and have it always work even when there is no user-specified message.

Because Chatty uses so few messages, we can assign descriptive names to them. In a larger program with many messages, it might be a better idea to use message numbers and default to printing out the message number when no message string is present.

Note that we did not limit ourselves to just messages. We allow the user to specify the font name, size and flags as well as the window title bar and the text for the speedup gadget. This technique can just as readily be used to construct menus and even requesters. The only limit to be aware of is that the tool types array is limited to 32K. A program which hits this limit should probably look for an alternative method of configuration (with that many strings, it is unlikely that a user would want to use the Info window for modifying them anyway).

The problems of proper multitasking and automatic sleep when deselected are both handled through the same mechanism. With a demo such as this, the common approach is to create a free running loop that checks for a message every so often. While this is not strictly a busy wait look, it often leads to performing an operation too often or wasting time waiting for something to happen.

One alternative is to use the timer device to regulate operations by waiting for a given interval between operations. However, there is a much simpler way to accomplish this. If you set the INTUITICKS IDCMP flag in the window, you will get a steady stream (about 30 a second or so) of event messages. These are not accurate enough for critical timing but do allow a program to do a little work every 'so often'.

For our program, we only update the screen when we get a message. With INTUITICKS, we get this constant flow of work at no performance degradation. One extra bonus is that when the window is not selected, we stop receiving the messages - allowing our application to effectively go to sleep.

Our speed control gadget illustrates how to create a gadget that automatically adjusts for the size of the window. In this case we simply want it to take the entire bottom border (excluding the sizing gadget). To do so, we set the GRELBOTTOM and GRELWIDTH flags and put in negative values for the *top* and *width* fields. The *top* value indicates how far from the bottom of the window to start the gadget, and the *width* field indicates how much space to leave when resizing the gadget.

The last thing this program illustrates is how to determine the size of a string. When printing out the random strings, the program determines the random location by calculating the box the string would fit in (through the code given above) and then finding a place in the window to put the string. In this way, unless the string is larger than the window width it should never write over the borders of the window.

### Day 6 - Heading home

As you can see from this short example, handling the issues of a well rounded program does not require a lot of code. The tricks and subroutines presented here should readily move to another program that is destined to see the world. All it really takes is being aware of the issues.

As a final note, I have deliberately not handled one example of internationalization in this program. Test your understanding by finding and fixing that problem.

**Solution:** (read no further if you want to take the challenge!)

The height for the gadget should be drawn from the font that the user specifies. By default, the gadget text is drawn in the default system font, but no provision is made to check the particular font asked for or even to look at the default system font. Simply add the lines:

```
speedtext.ITextFont = &mytextattr;
speed.Height = fontheight;
speed.TopEdge = -fontheight;
fontheight += fontheight-8;
 /* Account for extra space above size gadget */
```

Immediately after the line that says:

```
fontheight = font->tf_YSize;
```

```
/*************************************************************************/
/* CHATTY.C - Demo program to illustrate international issues          */
/* By John A. Toebes, VIII                                             */
/* Based on the original one published in:                             */
/*  The Amigan Apprentice and Journeyman Volume III, Number 2          */
/*  P.O. Box 411, Hatteras NC 27943                                    */
/*************************************************************************/
#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfxbase.h>
#include <workbench/workbench.h>
#include <workbench/startup.h>
#include <workbench/icon.h>
#include <libraries/dosextens.h>
#include <dos.h>
#ifdef MANX
#include <functions.h>
#define U_ARGS(a) ()  /* No support for function prototypes */
#else
#define U_ARGS(a) a   /* Function prototyping enabled */
#include <proto/exec.h>
#include <proto/intuition.h>
#include <proto/icon.h>
#include <proto/dos.h>
#define textAttr TextAttr
#include <proto/diskfont.h>
#include <string.h>
#endif

#define BASEROW 11  /* First Row we can write on */
#define BASECOL 4   /* First column we can write on */
#define EXTRAROW 20 /* Overhead for title bar, borders and sizing gadget */
#define EXTRACOL 8  /* Overhead for borders */

#define MAXSTRINGS  26
extern struct WBStartup *WBenchMsg;
struct IntuitionBase *IntuitionBase;
struct Library *DiskfontBase;
struct GfxBase *GfxBase;
struct Library *IconBase;
struct IntuiText speedtext = {
   1, 0, JAM1, 5, 0, NULL, "Go Fast", NULL};

struct Gadget speed = {
  NULL, 4, -8, -20, 8,   /* Next, left, top(v), width(v), height */
  GADGHCOMP|GRELBOTTOM|GRELWIDTH, TOGGLESELECT, /* flags, activation */
  BOOLGADGET, NULL, NULL, &speedtext, 0, /* type, render, select, text, mutex */
  NULL, 0, NULL };  /* specialinfo, gadgetID, Userdata */

struct NewWindow newwindow = {
  0, 0, 0, 0, /* left, top, width, height - filled in later */
  -1, -1,     /* Detail, block pens */
  NEWSIZE|CLOSEWINDOW|INTUITICKS, /* IDCMP flags */
  WINDOWSIZING|WINDOWDEPTH|WINDOWCLOSE|WINDOWDRAG|
  SIMPLE_REFRESH|ACTIVATE,
  &speed, NULL, NULL, NULL, NULL, /* gadgets, checkmark, title,screen, bitmap */
  30, 30, 99999, 99999, WBENCHSCREEN /* min/max width/height, type */
};

struct IntuiText itext = {
   0, 1, JAM1, 0, 0, NULL, NULL, NULL };

struct DiskObject *diskobj;
struct Library *DiskFontBase;
char *getstr U_ARGS((char **, char *, char *));
char **findtools U_ARGS((char *, BPTR));

#define M_CANTOPEN (getstr(toolptr, "OPENFAIL", "Can't Open the window"))
#define MSG(a) Write(Output(),a,strlen(a))

void main(argc, argv)
int argc;
char **argv;
{
struct Preferences prefs;
short len, rows, cols, x, y;
struct Window *window;
struct IntuiMessage *message;
static short modes[4] = { JAM1, JAM2, COMPLEMENT, INVERSVID };
char *(strings[MAXSTRINGS]);
int count, i;
struct WBArg *wbarg;
char **toolptr, *name, lookup[8];
struct TextAttr mytextattr;
struct TextFont *font = NULL;
int fontheight;

   /* Open all of our libraries that we need */
   if ((IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library",0)) == NULL)
   {  MSG("intuition.library?");   goto done;  }

   if ((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library",0)) == NULL)
   {  MSG("graphics.library?");    goto done;  }

   if ((IconBase = OpenLibrary("icon.library",0)) == NULL)
   {  MSG("icon.library?");        goto done;  }

   /* Note we don't care if this one fails.  We just won't change fonts */
   DiskfontBase = OpenLibrary("diskfont.library", 0);

   GetPrefs(&prefs, sizeof(struct Preferences));

   /* Now figure out where we were called from and get our icon file */
   if (argc) /* CLI */
      toolptr = findtools(argv[0],
         ((struct Process *)FindTask(NULL))->pr_CurrentDir);
   else /* Workbench */
   {
      wbarg = &(WBenchMsg->sm_ArgList[WBenchMsg->sm_NumArgs-1]);
      toolptr = findtools(wbarg->wa_Name, wbarg->wa_Lock);
   }
   /* See if they gave us a real title for the window */
   newwindow.Title = getstr(toolptr, "TITLE",  "No ICON file for TITLE");

   mytextattr.ta_Name  =      getstr(toolptr, "FONT",      "topaz");
   mytextattr.ta_YSize = atoi(getstr(toolptr, "FONTSIZE",  "9"));
   mytextattr.ta_Style = atoi(getstr(toolptr, "FONTSTYLE", "0"));
   mytextattr.ta_Flags = atoi(getstr(toolptr, "FONTFLAGS", "0"));

   speedtext.IText = getstr(toolptr, "GADGET", "Go Fast");

   if (DiskfontBase != NULL) /* Make sure we can open fonts safely */
      font = OpenDiskFont(&mytextattr);

   if (font != NULL)
   {
      itext.ITextFont = &mytextattr;
      fontheight = font->tf_YSize;
   }
   else
      fontheight = 9;  /* Assume something reasonable */

   /* Now go through and get strings for the table */
   strcpy(lookup, "STRINGA");
   count = 0;
```

```
for (i = 0; i < MAXSTRINGS; i++)
{
   if ((name = FindToolType(toolptr, lookup)) != NULL)
      strings[count++] = name;
   /* Advance to the next string to look for */
   lookup[6]++;  /* make STRINGA to STRINGB to STRINGC ... */
}

/* Default when we have no strings to something obvious */
if (count == 0)
   strings[count++] = "Unable to find message strings";

/* Calculate how big to open the window taking into account interlace and */
/* any morerows specifications that they have given                       */
rows = GfxBase->NormalDisplayRows;
cols = GfxBase->NormalDisplayColumns;
if (prefs.LaceWB)
   rows += rows;
newwindow.Width = cols;
newwindow.Height = rows;

if ((window = OpenWindow(&newwindow)) == NULL)
{
   MSG(M_CANTOPEN);
   goto done;
}
/* Account for the border overhead */
rows -= EXTRAROW + fontheight;
cols -= EXTRACOL;

for(;;)
{
   /* Wait for the next event to happen */
   while ((message = (struct IntuiMessage *)GetMsg(window->UserPort)) == NULL)
      Wait(1 << window->UserPort->mp_SigBit);

   switch(message->Class)
   {
      case CLOSEWINDOW:  goto done; /* They said go away ! */

      case NEWSIZE:      cols = window->Width-EXTRACOL;
                         rows = window->Height-(EXTRAROW+fontheight);
                         break;

      case INTUITICKS:   break;  /* This lets us process slowly */
   }

   /* Let intuition know we have processed the message */
   ReplyMsg((struct Message *)message);

   /* See how many times we need to do it */
   i = (speed.Flags & SELECTED) ? 5 : 1;

   while(i--)
   {
      /* Set up the string to be displayed (randomly) */
      itext.IText = strings[((rand() >> 2) % count)];
      itext.FrontPen = ((rand() >> 2) & 3);
      itext.BackPen  = ((rand() >> 2) & 3);
      itext.DrawMode = modes[((rand() >> 2) & 3)];

      /* Figure out where to display it safely */
      len = IntuiTextLength(&itext);
      x = BASECOL + ((rand() >> 2) % (cols - len));
      y = BASEROW + ((rand() >> 2) % rows);

      /* Display the string */
      PrintIText(window->RPort, &itext, x, y);
   }
}

done:
   /* Clean up everything */

   if (font)
      { CloseFont(font);                              font         = NULL; }

   if (DiskfontBase);
      { CloseLibrary(DiskfontBase);                   DiskfontBase = NULL; }

   if (diskobj)
      { FreeDiskObject(diskobj);                      diskobj      = NULL; }

   if (window)
      { CloseWindow(window);                          window       = NULL; }

   if (GfxBase)
      { CloseLibrary((struct Library *)GfxBase);      GfxBase      = NULL; }

   if (IntuitionBase)
      { CloseLibrary((struct Library *)IntuitionBase); IntuitionBase = NULL; }

   if (IconBase)
      { CloseLibrary(IconBase);                       IconBase = NULL;      }
}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Subroutine: findtools                                                     */
/* Purpose:    Locate a tools array for a given name                         */
/* Parameters: name - null terminated name of file to look for              */
/*             lock - directory to look for file relative to                 */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
char **findtools(name, lock)
char *name;
BPTR lock;
{
BPTR olddir;
char **tools = NULL;

   if (lock == NULL) return(NULL);
   olddir = CurrentDir(lock);
   if ((diskobj = GetDiskObject(name)) != NULL)
      tools = diskobj->do_ToolTypes;
   CurrentDir(olddir);
   return(tools);
}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Subroutine: getstr                                                        */
/* Purpose:    determine a string for a given handle                         */
/* Parameters: toolptr - pointer to tool type array to search                */
/*             name - handle of string to search for                         */
/*             deflt - default string to return if handle not found          */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
char *getstr(toolptr, name, deflt)
char **toolptr, *name, *deflt;
{
char *found;

   found = FindToolType(toolptr, name);
   return(found ? found : deflt);
}


#ifndef MANX
/* Tricks to make the code smaller */
void MemCleanup(){}
#endif
```

# Survival guide to the BridgeBoard

## *(or the Amiga 1000 with SideCar)*

**by Dan Schein**

*Dan Schein is a former Commodore Amiga Technical Support staffer who now works in the computing department of a Pennsylvania hospital. He has written on Amiga technical topics for several publications.*

With the addition of a BridgeBoard the Amiga 2000 becomes a multi-faceted machine. This combination is like no other machine before, or is it? Well to be quite honest, the answer is yes. The Amiga 2000 with BridgeBoard is functionally the same as an Amiga 1000 with SideCar. So even though I will be referencing the 2000 and BridgeBoard in the remainder of this article, all information will equally apply to the 1000 with SideCar. If you do have a 1000 with SideCar and find that you do not have some of the utilities and programs I refer to in this article, I suggest you stop by your local Amiga dealer and obtain a copy of the latest BridgeBoard software. Several new programs and options have been added since the release of the SideCar, not to mention several releases of SideCar software, but the important thing is that the latest version of BridgeBoard software will work with the SideCar and it is this version which is preferred.

The remainder of this article is based on two assumptions. The first is that you have followed the installation instructions in your BridgeBoard manual, and that your BridgeBoard has been tested and is in working order. The second is that you have a working knowledge of both MS-DOS and AmigaDOS. Items you will find useful are: the 2088 BridgeBoard manual, the MS-DOS 3.2 manual, your Amiga manual, a blank 3.5" disk, and a few spare hours for experimenting.

The newly-added power afforded by this marriage of operating systems raises a number of questions. The point of this article is to answer those questions by clearing up the most common areas of confusion: ways to transfer files between MS-DOS and AmigaDOS, setting up a Janus hard disk, virtual drives, and setting up the A1010 (3.5" drive) to format 720K.

### 720K on a 3.5"

Last thing first: getting 720K from the 1010. When you connect your 1010 to the BridgeBoard, it defaults to a formatted capacity of 360K. So why do we want this 720K format? Well, it is as much of a standard as the 360K 5.25" MS-DOS format. Most MS-DOS laptops use the 720K 3.5" format, and the new Big Blue models come with 3.5" drives that can read/write the 720K, format, so we want to be compatible. Even if none of this compatibility talk interests you, just remember that 720 is exactly double 360. Now *that* should be of interest to you.

Enough talking, let's double our storage and become compatible. All we need to do is add one line to the CONFIG.SYS file. The CONFIG.SYS file is a configuration file that MS-DOS checks at start-up. The CONFIG.SYS file must be located in the root directory of the disk you use to boot your BridgeBoard. (This means the MS-DOS boot disk and *not* the disk you use to start the PC Window software or Amiga). If your MS-DOS boot disk does *not* have a CONFIG.SYS file, you can use "edlin" to create one. If you *do* have a CONFIG.SYS file, you can still use edlin to add the new line to your existing file. Edlin is a MS-DOS line editor supplied by Commodore with your BridgeBoard. Directions for using edlin can be found in Part 2 - Chapter 6 of the MS-DOS 3.2 manual. Here is the line we want to add:

```
driveparm=/D:01 /F:2 /H:2 /S:9 /T:80
```

The spaces between the different flags are needed. Be careful and make sure you enter this line exactly as shown. Details of *driveparm* and what each item means and does, can be found in Appendix B of the MS-DOS 3.2 manual. Now we have to make the CONFIG.SYS file active. To do this we need to warm-start (reboot) the BridgeBoard. To reboot only the BridgeBoard, make sure the PC Window is active by clicking inside that window. Then hold down the ALT and CTRL keys while pressing the DEL key (decimal point) on the numeric keypad. Now your system consists of a 5.25" 360K drive, drive 'A' (this is the drive inside your 2000) and a 3.5" 720K drive, drive 'B' (this is the A1010 drive connected to the back of the BridgeBoard). How do you test if everything is working as planned? Easy! Just format a disk in drive 'B'. After formatting a disk in drive 'B', you should see this on your screen:

```
730112 Bytes total disk space
730112 Bytes available on disk
```

If you don't, go back and check your CONFIG.SYS file.

(Sidenote: If you are using two 5.25" (A1020) drives on your BridgeBoard, this CONFIG.SYS trick will also work for you. But be aware that formatting 720K on a 5.25" disk will result in a disk that most MS-DOS users will not be able to read.)

**Virtual drives**

The next item we will be covering is virtual drives. Virtual drives are covered in the BridgeBoard manual but require some clarification and further explanation. I'll give an example of how to set up a virtual drive but I suggest that you also take the time to read Appendix C of the BridgeBoard manual. There are three steps you must take before you can use a virtual drive. Step one is to add this line to your CONFIG.SYS file:

```
device=jdisk.sys
```

Now that you have edited and saved the CONFIG.SYS file, let's reboot the BridgeBoard and move on to step two. It's important to remember that step one (editing the CONFIG.SYS file) must only be done one time, while steps two and three must be done *every* time you plan on using a virtual drive.

*The BridgeBoard begins to boot when the Amiga 'binddrivers' command is executed, whether you open the PC Window or not...*

Step two consists of running the "PCdisk" program. The PCdisk program can be found in the PC Drawer of the Amiga workbench screen (disk). To run PCdisk simply double click on the PCdisk Icon. We now have a two-way communication channel set up between MS-DOS and AmigaDOS. For this example we will be using the RAM disk on the AmigaDOS side to hold our virtual drive, but any drive (floppy or hard) could be substituted.

Step three, creating the actual virtual drive. Type in the following on the MS-DOS screen:

```
jlink d: ram:ms-dos /c:160
```

Now press Return. The display should now show:

```
VDrive    Status    Linked to
--------------------------
c:
d:        R/W       ms-dos
e:
f:
```

We've now created a file called "ms-dos" in the Amiga's RAM disk (RAM:). This file is linked to the MS-DOS side via the drive name 'D'. This new file can be seen if we do a directory of the RAMdisk. It's important to recognize that this file cannot be accessed, copied, or deleted by AmigaDOS until the file is closed. (I will explain the process of closing a file later). However, we can now use (access) drive 'D' just like any other MS-DOS drive, let's try it!

Do a directory of drive 'D'. Copy your CONFIG.SYS file to drive 'D'. Now type the CONFIG.SYS file. After typing the file, delete it and do a directory again. As we have just demonstrated, a virtual drive is used exactly the same as any other drive . (If you do not know how to do a directory, or copy or type a file, refer to the MS-DOS 3.2 manual for assistance).

Now that we have a virtual drive and know how to use it, I'll explain what the different options on the jlink line mean. The "d:" means that we'll refer to the virtual drive as drive 'D'. The "ram:ms-dos" indicates that we want our virtual drive stored in the Amiga's "RAM:" with the name "ms-dos". The "/c:160" defines the file with a maximum size of 160KB. If we had not used the "/c" option and the file "ms-dos" did not exist, the jlink command would not have succeeded and an error message would have been returned. I used the number 160 because this is the smallest size you can specify for a virtual drive. (And you thought I just made it up, didn't you)? If you use a number smaller than 160, the size is automatically set to 160KB. The 160KB size is not the actual size of the virtual drive, but rather the maximum size that the virtual drive can become. Judging by the looks on your faces, I guess I'd better explain that one....

A virtual drive works very much like our Amiga's RAMdisk in the sense that it is never bigger than needed but will expand as required (as long as we don't try to exceed the file size that we selected in the jlink command above). What this means is that as we put more files in our virtual drive it will expand accordingly; however, unlike our Amiga's RAMdisk, when we delete files it will stay at its present size. A virtual drive never gets smaller, only bigger.

Two other important items need to be discussed: unlinking and relinking of virtual drives. A linked virtual drive should be unlinked after you are finished using it, and before you power off. If you were to power off without unlinking a virtual drive, the file (virtual drive) might be damaged and you will have problems linking this file the next time you try. Let's unlink our virtual drive. Enter this line:

```
jlink d: /u
```

Now press Return. Your display should now show:

```
VDrive    Status    Linked to
--------------------------
c:
d:
e:
f:
```

This indicates that our drive has been unlinked successfully and that no drives are linked at this time.

(Sidenote: A word of caution about automatically linking a virtual drive at start-up (in your AUTOEXEC.BAT file). The BridgeBoard begins to boot (provided a boot disk is in drive 'A') when the Amiga 'binddrivers' command is executed whether you open the PC Window or not. This means that if you have your virtual drive linked automatically at power up and you power off without starting a PC Window and unlinking, you most likely will damage your virtual drive. I hope those files weren't important! There are three ways to prevent this accident . The first is to not use the jlink command in your start-up script (AUTOEXEC.BAT). The second is to always open a PC Window and unlink before powering off. Easier said than remembered. The third (my personal choice) is to install the "Date" (or "Time") command in AUTOEXEC.BAT *before* the jlink command. This means that if you don't open a PC Window and enter the date, you have no worries of losing your virtual drive and your data because the jlink command was never executed. One important fact, which all this auto-linking discussion is based on, is that you run the PCdisk program either manually or as part of a start-up-sequence.)

Relinking a file is easy. Let's relink our virtual drive that we have been using for all our tests. Enter this line:

```
jlink d: ram:ms-dos
```

Now press Return. Your display should now show:

```
VDrive    Status    Linked to
---------------------------
c:
d:        R/W       ms-dos
e:
f:
```

As you can see, our virtual drive is back and ready for use. Notice that we did not use the "/c" option this time. If we had, any of the data in our old file would have been lost.

If you enter the "jlink" command alone with no additional parameters, jlink will list the drive names available for use as virtual drives and the status of any active virtual drives. You may have noticed by now that each time we use the jlink command it displays a list of four drive names. This is because you are allowed a total of four virtual drives. Any of the displayed drive names can be used and they do not have to be used in any specific order. Your jlink display will show different drive letters if you have more or less than two drives. The list displayed by jlink always starts at the next available device letter.

Let me finish with virtual drives by mentioning the "/r" option. This option - if used when creating (or relinking) a virtual drive - will create a read only device. All write attempts will fail and be greeted by a MS-DOS warning message.

Congratulations! You have survived Virtual Drive 101, the introduction course. Time to move on to our next subject.

**Janus drives**

A Janus drive is actually an AmigaDOS partition residing on an MS-DOS hard drive that the Amiga accesses as if it were an Amiga device. MS-DOS allows multiple partitions on its hard disks. With this handy feature and some utilities supplied by Commodore, we can turn one of these partitions into a Janus drive.

Due to the large number of hard drive, controller, and hard drive card manufacturers, we will *not* explain how to install these items. The hard drive in this case is a 20MB hard disk card that will be partitioned into two evenly sized partitions. Adjustments should be made for different size hard drives and/or partition sizes as required.

*A Janus drive is actually an AmigaDOS partition residing on an MS-DOS hard drive...*

The BridgeBoard, like any other MS-DOS based computer, requires you to run the "Fdisk" program before formatting the hard disk. Fdisk can be found on the MS-DOS system (boot) disk supplied with your BridgeBoard. Fdisk is explained, in great detail, in Part 3 - Appendix F of the MS-DOS 3.2 manual. Rather than rewrite all this information, I will simply give enough information to use Fdisk to create the partition for our Janus drive. For more information, refer to the MS-DOS 3.2 manual.

Enter "Fdisk" and press Return. After the menu appears, select option one. Since we want to use part of this drive as a Janus drive, enter "n" when asked if you want to use the entire hard disk. Now we are shown the total number of cylinders available for use (611), and prompted to enter the size of our partition. Since we want to divide our hard drive in half, enter 312 and press Return. Next we are prompted for the starting cylinder number, enter 0 and press Return. Fdisk will now inform us that our DOS (MS-DOS) partition was created. Next, select option two. We are requested to enter the number of the partition we want to make active. Enter '1' and press Return. That's it - we're done using Fdisk. Use the escape key to exit. After exiting Fdisk, you must reboot the BridgeBoard.

After your BridgeBoard has rebooted, let's partition for the Janus drive. Enter "Adisk" and press Return. You are now presented with options and information that should look similar to those in Fdisk. This information tells us that there are currently no Amiga drives partitioned, that the total hard drive size is 611 cylinders and that there is an existing MS-DOS partition active. This MS-DOS partition starts at cylinder 0 and ends at cylinder 311. Select option three. We are now shown the number of cylinders available (298) and the first available starting cylinder (312). Let's enter 298 and 312, Adisk now in-

forms us that partition number two is an Amiga partition starting at cylinder 312 and ending at cylinder 611. This gives us 298 cylinders (not quite 50/50, but close enough). Use the escape key to exit Adisk, and then reboot your Amiga.

The next thing we will do is format both partitions of our hard drive. Open the PC Window and let's start with the MS-DOS side. Enter the following line:

```
format c: /v /s
```

Press Return. We are now presented with a warning that all data on the non-removable disk (i.e. the hard disk) will be lost. Just enter "y" and press Return. The system will now begin to format the MS-DOS partition of our hard drive. When formatting is finished we will be prompted for a volume label. The volume label is similar to the disk names (labels) we give our Amiga disks. Enter the name of your choice and press Return when finished. Your MS-DOS partition is now formatted and ready to use. If you are not familiar with copying files, refer to the MS-DOS 3.2 manual, Part 2 - Chapter 3 (Users Guide).

Now let's switch to the AmigaDOS side and open a CLI window. If you are not familar with the CLI, refer to your Amiga manual. Type "DJmount" and press Return. After a few seconds a requester should appear telling you that your hard disk is not a DOS (AmigaDOS) disk. Just ignore this message and click on cancel. Enter the following:

```
DPformat drive jh0: name "Hi Mom!"
```

Replace "Hi Mom!" with the name you want for your hard drive. Now press Return. We are requested to insert a disk. Since the disk is already in place, we can ignore this message and press Return. The system will now format our AmigaDOS partition (Janus drive). When formatting is finished, the CLI prompt will return and a new icon for our Janus drive will appear on the Workbench.

All that's left is to have our Janus drive mount automatically at start-up. From the CLI enter:

```
ed df0:s/startup-sequence
```

Place a *copy* of your Workbench (boot) disk in drive df0: and press Return. After the ED window opens, move the cursor so that it is directly to the right of the "s" in the line containing the command "binddrivers". Press Return. A new line will open and your cursor will be on it. Enter "DJmount" and press the escape key. An asterisk (*) appears at the bottom of the screen. Type "x" and press Return.

With the DJmount command as part of the startup-sequence, the Janus drive will mount at start-up automatically. It will also cause a delay of about 1.5 minutes while it mounts.

That's all there is to Janus drives. By now you should be beginning to feel like a BridgeBoard expert!

**Transferring Files**

The last part of this article will cover the many ways available to transfer files between MS-DOS and AmigaDOS. There are several utilities available to help you transfer files. Some are provided with the Amiga, some with the BridgeBoard, and others are commercial programs available through your local Amiga dealer.

One important thing to remember is that just because you can copy a binary (program) file from one DOS type to another, it does not mean that that program will run correctly - or even at all. No conversions are done on transfers - these utilities are primarily designed for transferring text files.

Let's start with the PCutils. These utilities were first released by Commodore as part of AmigaDOS V1.2 (The Enhancer Kit). They are now included with every Amiga sold. The PCutils consist of two programs: PC Format, and PC Copy. The PCutils will work with MS-DOS 360K 5.25" formatted disks and the Commodore A1020 (5.25") drive only. Note that the 1020 drive may *not* be mounted (i.e. made usable by AmigaDOS) when using the PCutils.

PC Format allows you to format a MS-DOS disk in your 1020 drive. Very handy if you need a MS-DOS disk because you sat down to transfer files and forgot to format a disk. Using PC Format is very straightforward. After clicking on the PC Format icon, a window will open with all the options preset. You can change any option or enter a volume name as required. Normally you will use it without altering the preset options.

PC Copy is the real workhorse of the PCutils pair. PC Copy can copy text or binary files. There are two special options when copying text files. Text-7 and Text-8 are 'filtering gadgets' that ensure text files are converted correctly for use on the opposite DOS type. PC Copy can be used to read and write to both MS-DOS and AmigaDOS.

When the PC Copy program is run, the options on the displayed window are set for copying binary type files from the 1020 to your Amiga. If you are going to copy text files, set the filtering option to the correct setting first. If you did not have a disk in the 1020 drive when you started PC Copy or you changed disks in the 1020 drive, click on the word "Disk Change" in the lower left corner to inform PC Copy. If the MS-DOS disk you are using has hidden files, clicking on the word "Hidden" at the bottom center of the screen will cause them to be displayed. If you wish to copy from AmigaDOS to MS-DOS, click in the "From: - To:" window. This will switch the source and destination drives. When you're copying from AmigaDOS to MS-DOS, you also have three additional options. These options are controlled via a pull down menu. These options include Read-Only, Hidden, and System. If you are not familar with exactly what these items mean, refer to the MS-DOS 3.2 manual. That's it for PCutils, so if you have a 1020 drive - give them a try. They may not be the very best, but they sure are some of the best priced.

Next are the ARread and AWrite utilities supplied on the MS-DOS disk with your BridgeBoard. An important thing to note is that in order for ARead and AWrite to work, both require that the PCdisk program be run first. (See the section on virtual drives in this article for information on the PCdisk program.)

ARead copies an AmigaDOS file over to a MS-DOS file. The AmigaDOS file may reside on any valid AmigaDOS device. Examples include floppy drives, hard drive(s), and RAM drive. ARead can write the file to any valid MS-DOS device. The syntax for ARead is:

```
ARead df0:s/startup-sequence A:test.txt
```

This will copy the Amiga file "startup-sequence" from the "s" directory of drive "df0:" over to the MS-DOS drive "A:" and name it "test.txt". One item Commodore forgot in the BridgeBoard manual is that if the file you are going to copy is binary, you'll need the "/B" switch on the end of the command line. Here is an example:

```
ARead df0:c/echo A:echo.exe /B
```

That's it for ARead, now let's talk about AWrite. AWrite does the opposite of ARead. It copies a MS-DOS file over to an AmigaDOS file. Like ARead, any valid device can be used. Here is the syntax for AWrite:

```
AWrite A:autoexec.bat
    df0:testfile
```

This reads the "autoexec.bat" file from the MS-DOS drive "A:" and writes it to a file called "testfile" on the AmigaDOS drive "df0:". As with ARead, the "/B" switch should be used when copying binary files.

*One item Commodore forgot in the BridgeBoard manual is that if the file you are going to copy is binary, you'll need the "/B" switch on the end of the command line...*

An interesting use of AWrite is to copy MS-DOS text files to your Amiga's printer. The syntax is:

```
AWrite A:autoexec.bat prt:
```

There are advantages to using AWrite in place of the "LPT1" program. These advantages include using Preferences to control the type of print output and the type of printer you are using. (Now you can use a serial printer or any other printer that the Amiga can use!)

Now let's take a look at Dos-2-Dos. Dos-2-Dos is a commercial program that, like PC Copy, runs under AmigaDOS and can use the 1020 drive. This is where the likeness ends. Dos-2-Dos can read and write 3.5" 720K and 3.5" 360K MS-DOS formatted disks in addition to 5.25" 360K disks.

Dos-2-Dos is a very easy program to use, but it can only be run from the CLI. I will not cover how to use Dos-2-Dos, since that is amply explained in the manual that comes with the program. The main advantages of Dos-2-Dos over PC Copy are: any drive can be used, even if it is mounted; and you also have the ability to read and write 3.5" MS-DOS disks. If you find these items to be desirable, consider Dos-2-Dos as an addition to your toolbox.

(Sidenote: Dos-2-Dos has one other very interesting ability. It can read and write Atari ST formatted disks as well as MS-DOS and AmigaDOS.)

The final program I will mention in this roundup of conversion tools is called "Project D". Project D is an Amiga backup utility with a new twist called Omni Copy. Omni Copy is unlike the previous utilities I have mentioned. Specifically, it does *not* copy between different operating systems. Omni Copy copies disks of the same DOS; however, various drive types (3.5" and 5.25") are supported, with a provision for multiple destination drives.

To understand the powers of Omni Copy, consider this situation. You have an Amiga with a 1020 drive, and your friend just bought a new 500 and Transformer. He wants to run Transformer on his 500 but has no 3.5" MS-DOS boot disk. He does own a PC clone so he has a 5.25" MS-DOS boot disk. He brings a blank 3.5" disk and his 5.25" MS-DOS boot disk over to your place. Using Omni Copy, you transfer his 5.25" MS-DOS boot disk into a 3.5" MS-DOS boot disk. With one easy step, disks can be converted from 5.25" to 3.5".

Of course Omni Copy also copies from 3.5" to 3.5" or 3.5" to 5.25" formats. This gives Omni Copy the ability to backup 3.5" disks or transfer from 3.5" to 5.25" (or vice versa) all with one program. Considering the fact that Project D is primarily marketed as an Amiga backup program, the additional powers and versatility of Omni Copy make this a good value.

Sidenote - In addition to MS-DOS formats, Omni Copy can also copy Atari ST, CP/M, and Xenix formatted disks as well.

**Out of the woods**

Well you have reached the end of of your Survival Guide to the BridgeBoard. With all this newly obtained knowledge, you will hopefully run into fewer problems, and also find the BridgeBoard to be a rewarding experence.

There are many resources available to PC users, and we will not be including PC coverage in future issues. However, if you feel there is a need for more articles specific to the Amiga/BridgeBoard environment and you have an idea for an area you would like to see covered, send your requests to me via *Transactor for the Amiga*.

# Manx C 3.6 and *sdb*

## *It should get better, but it's good now*

**Review by Nick Sullivan**

Manx Aztec 3.6a C Development System
(Developer Version): $299 US
Upgrade from version 3.4: $30 US (or free with *sdb*)
*sdb* Source Level Debugger: $75 US
Both from Manx Software Systems
(800) 221-0440; in NJ or outside the US call (201) 542-2121.

Version 3.6 of the Manx Aztec C compiler package is in most ways not a major upgrade from the previous release, version 3.4. In fact, the performance of the main component programs (compiler, assembler and linker) seems about the same, both in execution speed and in the size and speed of the generated code. Manx has been working on an ANSI-conformant compiler, and on design improvements that would allow better code generation, but the changes here are not so radical.

What *is* new in 3.6 is support for *sdb*, Manx's long-awaited source level debugger. As you'll see later, *sdb* needs to know a lot about the source files in the program you're debugging. It gets this information from a special ".dbg" file generated by the linker (via a command line switch). The linker, in turn, can extract the debugging information from the object (".o") files, provided the compiler was also invoked with the correct switch. The implementation of these switches is the only really significant innovation in the 3.6 release.

The other changes are minor upgrades. Bugs from the 3.4 version have been repaired, and a few new options have been added. The most interesting is the capability of generating in-line code for the strcpy(), strcmp() and strlen() functions, either in all instances (with a compiler switch) or individually, by preceding the function name with the string "_BUILTIN_".

But, as I said, this version of the compiler exists primarily to support *sdb*, and *that* product is of more than passing interest.

### Why a source level debugger?

Anyone who has used a conventional machine language level debugger (such as Manx's *db* or Metadigm's Metascope) to debug programs coded in C will recognize the need for a program specialized for debugging at the source level. Conventional debuggers have access only to those symbols that are global in the source program; in C, that means the debugger knows the names of variables and functions whose storage class is external, but can reference others only by an absolute address, an offset from a global or register pointer, or a register name. Hence you waste a lot of time calculating addresses when you want to examine the contents of a static, automatic or register variable, or disassemble a static function.

There are other frustrations too. References to data structures in a C program are presented to you as numeric offsets from the structure base by a conventional debugger, not by the name of the structure member. (A0 holds a pointer to a Window structure; the instruction is "move.l 52(a0),d0"; quick - what structure member are we looking at? If you said "the IDCMP Flags", you probably don't need a source level debugger. You probably *do* need a social life.) Moreover, single-stepping through a C program at the machine level can be very enlightening, but for practical debugging it isn't very efficient.

Source level debugging provides an answer to all these problems. If you want to know the current contents of an auto variable, you can ask for it by name. If you want to examine that Window structure, you can get a complete display with all the member names, their types and their present values. And you can single step by *source* instructions, and forget about machine language altogether.

### Breakpoints

Single-stepping is equivalent to setting a breakpoint in your program after each instruction. That isn't very flexible. You usually begin a debugging session with a rough idea of the problem area in your code; you may want to get to that area quickly and single-step through it. Often you'll just want to pause at one or a few places in the program, and examine the contents of variables and other memory at those points (perhaps on entry to and return from a suspect function). In these situations, you need to be able to set arbitrary breakpoints, and let the program run full-speed until it hits one.

*sdb* offers breakpoints in a couple of flavours, which modify the conditions under which the breakpoint will be 'taken'. A simple breakpoint will halt the program and return control to you whenever the break address is reached; you specify the address as a memory location, a function name, a source line (in any of the source files from which your program was created), or a C expression yielding an address. You can optionally set a 'skip count' - the number of times the breakpoint must be reached before execution is halted there. You can also provide

a series of *sdb* commands - to display the contents of key variables, for instance - that should be executed automatically when the breakpoint is taken. Another form is the 'expression' breakpoint, which allows you to specify an arbitrary C expression to be evaluated on entry and exit to every function (or after each instruction when single-stepping). Execution is halted if the expression returns a non-zero result.

**A matter of interpretation**

The ability to understand C expressions entered at debugging time is extremely useful. Apart from its application with breakpoints and a couple of other commands, this feature may be employed directly to evaluate an expression and return its result. Since function calls and assignments are legitimate C expressions (or sub-expressions), you can manually call any function known to your program with various test inputs, or determine or modify the contents of variables. This is not quite interpreted C, but it certainly provides a lot of the convenience that a C interpreter would give you.

For simply displaying variables, though, a formatted print command is supplied. Here is an instance of its use:

```
>p *window->RPort->BitMap
struct BitMap = {
unsigned int BytesPerRow = 80
unsigned int Rows = 200
unsigned char Flags = 0
unsigned char Depth = 2
unsigned int pad = 0
unsigned char *Planes[0] = 0x00002e68
}
```

For determining quickly whether a structure in your program has been properly initialized, this is about as convenient a method as you'll ever find.

***sdb* and the stack**

C uses the stack for three main purposes: as storage for auto variables, to pass function parameters, and to save return addresses for function calls. Since the location of all these items is known or can be deduced given a knowledge of how the compiler uses the stack, *sdb* is able to provide a 'stack backtrace' from any point in the program showing the nested function calls and their parameters all the way back to _main(), and can show you not just the auto variables in the current function, but in all the partially executed functions en route to the current one. This information is often helpful in making sense of complicated situations - when dealing with functions that call themselves recursively, for example.

**Son of *db***

Users of Manx's machine level debugger, *db*, will notice more than a passing resemblance between it and *sdb*, right down to the command syntax. In fact, *sdb* incorporates much of *db*'s functionality, providing extensive facilities for debugging at the machine as well as the source level. For instance, the user can switch freely between single-stepping by source instruction and single-stepping by machine instruction. Commands to disassemble memory, to dump and modify memory and registers, to display Amiga-specific information such as the task and device lists, to create command macros, and to log the debugger's output, are all taken directly from *db*. Many of the breakpoint commands are identical or nearly so in both syntax and operation, and even the stack backtrace and formatted display commands lean heavily on precedents established by *db*.

The differences are as interesting as the similarities. Most obvious is that *sdb* is much more Amigacized - though in a somewhat quirky way. The program runs in a single window subdivided into three areas: an area on top for displaying source code; a full-width string gadget (not Intuition's, but a hand-rolled one) for user input below that; and an output area at the bottom. The relative size of the source and output areas can be adjusted by sliding the string gadget up and down, either from the keyboard or with gadgets. The window as a whole is also resizable, and the colours in each area are separately selectable. This is an unconventional set-up (by comparison with Metascope's multiple window display, for example), but it has the advantage that you can get the debugger completely out of the way by clicking a single window to the back.

Despite the window, the gadgets and the colours, *sdb* has a long way to go in the user interface department. All commands must be given as (more or less) mnemonic codes: 'bs' to set a breakpoint, 'acs' to set the colours in the source area, and so on. There are more than sixty of these codes, many of them accepting multiple argument forms; it may be a long time before you can get by without the manual. Considering the Amiga's flexibility in user input, it should not be difficult for Manx to make future releases of the program easier to learn and to control, and I hope they do so. For example, it would be handy if you could click on a line in the source area to set a breakpoint.

An important difference between *db* and *sdb* is that the latter does not live in the background, independent of the program being debugged. As a result, you cannot debug sub-tasks the way you can with *db* (although a special mode is available for debugging user-written libraries and devices).

**Documentation**

The manual for *sdb* comes in a small 3-ring binder, and amounts to little more than a detailed reference card. Almost nothing is offered in the way of tutorial help or examples (though a fairly extensive demo on the disk partly compensates for that), and some command descriptions are confusing, inaccurate or both - it's sometimes a little difficult to say. The 'p' command for formatted output is the most baroque of *sdb*'s commands, requiring nine pages of compact description to detail its many options. As far as I could tell after poring over the manual and experimenting with the program, most of the options don't work. Or perhaps I just didn't understand.

**Notwithstanding which...**

Anyone who does a lot of programming with the Manx C compiler will probably get more than enough use out of *sdb* to justify its purchase.

# ARexx

## *The REXX language comes to the Amiga*

**Review by Larry Phillips**

"It slices! It dices! It cuts through nails or overripe tomatoes with equal ease! But wait! It's also a floor polish and a lawn fertilizer!"

I'm sure you've all heard and hated the TV huckster pushing the latest gizmo that does so much and costs so little. I know I have, and I sometimes wonder if I sound like that when someone asks me what ARexx is all about for, as you will see, ARexx is multi-purpose tool that is difficult to praise too strongly.

There are a number of facets to ARexx that stand out. As a script language similar to, but far more powerful than, the AmigaDOS script language called by Execute, it provides an astonishing degree of control over system commands. From its powerful DO statement, which allows controlled looping, to the ability to parse any sort of input, whether it be files and their contents, directory lists, or keyboard entry, ARexx smacks of power and professionalism at every turn. Add to this the inter-process communication facility, which allows an ARexx program to interact either with another ARexx program, or with any program that has an ARexx interface written into it, and you can see why it is that at this point I am tempted to ask, "Now how much would you expect to pay? But WAIT! There's more!".

ARexx comes with two libraries, fully documented, that may be used with any program. They provide support for ARexx functions as well as more general, high-level functions. And finally (if I haven't forgotten something), ARexx supports external function libraries so that you can extend the language yourself.

First, let's look at the script capabilities of ARexx as compared with the standard AmigaDOS scripts. Here is a list of capabilities included with ARexx that can be implemented only with difficulty, if at all, in an AmigaDOS script.

- **Looping:** Looping is performed in one of two ways, either by jumping to a label or via a loop construct. The loop constructs are extremely flexible, allowing fixed counts, conditional exit, or even no exits at all.

- **Strings:** ARexx allows you to manipulate strings directly. The strings may be defined within the program, or may come from the user via keyboard, from a file, or as the result of a command. Strings may be split, concatenated, parsed, or manipulated, as pure strings or as lists of words, in almost every conceivable manner.

- **Selection:** ARexx provides a *select* keyword for the control of program flow. It is much like a series of IF/THEN statements, and has a default case for when none of the explicit cases match. Unlike the equivalent statements in some other languages, *select* allows any conditional statements for the actual tests.

- **Calculation:** Calculations may be performed on integer, floating point, or boolean values. ARexx is an 'untyped language', in that you may perform a string operation or a mathematical operation on the same variable, the success of which is only dependent upon whether or not the data item in question is suitable for the operation performed.

- **Procedures:** Procedures are implemented in much the same way as are procedures in other languages. They are subprograms with their own separate stack space and variables. Variables may be left hidden within a procedure, or they may be 'exposed' so that a procedure can get values from or change the value of variables external to itself.

- **Communication:** An ARexx program may communicate with other ARexx programs, or with any program suitably written in any other language. This facility may prove to be the single most important feature of ARexx. More about why later.

- **Tracing:** ARexx provides very powerful built-in tracing functions. Tracing may be turned on and off within an ARexx program, and may performed on commands, errors, intermediate results, labels, final results, or all of these. There is a 'scan' mode, whereby all clauses are traced without anything actually being executed, and an 'interactive trace' mode that is very handy for stepping through a program or to test ARexx statements directly from user-supplied keyboard input.

• **Interrupts:** ARexx allows you to set up actions to be performed upon receipt of various interrupts, such as CTRL C, D, E, or F signals, errors, uninitialized variables, etc.

• **Clipboard:** ARexx fully supports a clipboard (not the standard Amiga clipboard, but very usful nonetheless). This clipboard may be used to pass strings between programs, and even used as a place to store ARexx programs for execution by another ARexx program.

• **Interpret:** This ability is really one of the more esoteric of the features of ARexx. It allows ARexx to execute a string as if it was a program. See 'calc.rexx' later in this article for a practical example of its use in what is perhaps the shortest useful ARexx program.

It might be interesting at this time to look at a few simple ARexx scripts. (I should mention that I load the 'rexxsupport.library' during my startup-sequence to avoid having to explicitly check for and load it within each program that needs it. I also use WShell, another excellent product from William Hawes that allows me to call ARexx programs without having to type 'rx' in front of them. If you are not running WShell, you will need to use the 'rx' in the same way that AmigaDOS script programs require the 'execute' keyword - at least under AmigaDOS 1.2.)

The first is one that is included with the ARexx package, and shows the use of the *interpret* keyword. It is invoked with the command 'calc <expression>', where <expression> is any expression that may be evaluated by ARexx. It is a very handy little calculator and gets a lot of use around my computer room. Here's what it looks like:

```
/* calc.rexx */
interpret 'say' arg(1)
```

The first thing in any ARexx program is a comment. ARexx uses this comment to verify that it is a valid program. The comment and body of this program could have been typed on one line. The second line is the one that does all the work. It builds a string consisting of the word 'say', followed by a space, followed by the arguments passed to the program as <expression>. The *interpret* command then executes this string as if it were an ARexx program statement. Assuming I called this program by typing 'calc 6*5', the string built would be 'say 6*5', a valid ARexx statement ('say' is the ARexx way of printing to the standard output, or screen), and the result would be that it would print '30' on the screen.

I mentioned that <expression> could be any valid ARexx expression, which allows us to do some things with our first example that might not be immediately apparent. For instance I could type 'calc c2d('a5'x)'. This would print '165' on my screen, which is the result of the conversion of hexadecimal 'A5' to decimal. Another interesting (but not too useful) example might be 'calc words('this is weird')', which will print '3', the number of words in the string 'this is weird'.

How about something a little more useful? Here's a small script that allows you to use an '*' as a wildcard for the AmigaDOS COPY command:

```
/* copy -- adds * wildcard to copy command */
parse arg x
asterisk = pos('*',x)
do while asterisk > 0
   x = delstr(x,asterisk,1)
   x = insert('#?',x,asterisk-1,2)
   asterisk = pos('*',x)
end
c:copy x
```

Again, we have a comment as the first thing in the program. 'parse arg x' transfers all the arguments given on the command line to the variable 'x'. Note that we do not have to declare *x* as a variable, and we do not have to tell ARexx what kind of variable it is. The line 'asterisk = pos('*',x)' simply assigns a number to the variable 'asterisk' corresponding to the position within *x* of the first '*' it contains. If *x* contains no '*' character, asterisk will contain the value of 0.

The 'do' loop will execute if the value of 'asterisk' is nonzero, and will continue to do so as long as there are any '*' characters in *x*. We remove the first occurrence of the '*' character with the line 'x = delstr(x,asterisk,1)', and replace it with the standard AmigaDOS '#?' wildcard in the next line. We keep checking for more asterisks, replacing any we find, and when there are no more to replace, call c:copy with the modified command line arguments contained in the *x* variable.

**Programs that converse**

It's easy to get excited about the possibilities of using ARexx as a scripting language, easy to let your imagination run wild with possible scripts to make life easy for you, yet the most exciting part of ARexx lies in its ability to communicate between processes. ARexx programs may communicate with each other, which means that you can start up ARexx programs that open a named port and await the receipt of a message. When a message is sent to the port from another ARexx program, it can awaken, parse the message, and perform whatever action the programmer wanted from it.

Any program, regardless of the language in which it's written, may also have the ability to understand ARexx messages. Writing an ARexx interface into a program is fairly simple, and allows a degree of flexibility that is hard to ignore. There are currently two editors, TxEd Plus and Uedit, that understand ARexx communications, and another that is due to arrive soon, perhaps even by the time you read this.

To fully appreciate what can be done with a program like TxEd Plus coupled with ARexx, consider that TxEd Plus can be commanded through its ARexx interface to do virtually anything you can do from the keyboard, and can pass various data back to ARexx. These data can be the contents of the cur-

rent line, the title of the file being edited, and much, much more.

Have you ever admired Turbo Pascal's user interface, which allows you to write a program, compile, link, debug, and test run it without ever leaving the editor? I know I have, and now it can be easily done with ARexx, TxEd Plus, and your favourite compiler. An ARexx program controls the entire process, waiting for your signal through a user-defined menu or command key that sends a message to compile the program you have in the editor. The ARexx program might command Txed Plus to save the program, call the compiler, and report back the success or failure of the compile. If there were errors, it could read the error file and parse it in order to tell you what the errors are, and position your cursor on each offending line in turn. If the compile was successful, you could, with another menu selection, do a test run of the program. So, unlike Turbo Pascal, you have your choice of editor and compiler for this application.

As you can see, your editor becomes more than just a program for entering text. It becomes whatever you want it to be: an intelligent file reader or a tool for reading BBS messages, complete with the ability to let you answer them for later uploading, discarding some, keeping others in an archive file. Let your imagination run free and think of ways to use a powerful tool like this, and you'll see why I get a little excited when I tell people about ARexx.

Another good example is PCLO Plus, from SoftCircuits. PCLO Plus is a printed circuit board layout program, a 'big brother' to the original PCLO. The recent addition of an ARexx interface has elevated this already fine program to heights other machine owners can only dream of. Drill tape output, updating parts lists, even updating schematics from changes made while laying out the board are all possible.

It doesn't stop there. More and more applications are being fitted with an ARexx interface. Programs available with an ARexx interface now are AmigaTeX (a typesetting program), PCLO Plus, TxEd, Uedit, and probably more that I am unaware of. Microfiche Filer, a unique 'filing system', and an as yet unnanounced database manager with ARexx interfaces are 'coming soon'. Rumour has it that Access!, a shareware terminal program, may get an Arexx interface.

The most exciting part of all this is that you, the end user, can determine what programs you want to use together, gathering tools to fit your needs, rather than having to rely on the author of a program to anticipate your wishes.

I'm going to close off the discussion of ARexx as a communication medium between programs with some thoughts about something I have recently been considering. Suppose we had a 'very dumb terminal' program that didn't have file transfer protocols built in, no dedicated script language, telephone directory, file requesters, gadgets, and so on. Now imagine that it could, under control of an ARexx program, have user-definable menus that, when selected by the user, would be reported back to the ARexx program. It might have the ability to watch for and report user-definable strings as they are received. A crucial ability would be that it could be commanded to stop 'listening' to the serial port. Now what if we had a file requester, a couple of protocol programs that knew how to upload and download, and a 'gadget box', all of which could communicate with an ARexx program?

One can easily imagine using all these modules to build a terminal emulator with a lot of features, all glued together with an ARexx program, but there's more. If your favourite protocol isn't included, or you don't like the file requester or the gadget box, you can always write your own or convince someone else to write one. (There is already a file requester with an ARexx interface that is freely distributable).

Now for the best part: why be content with just a terminal program? The same set of modules used in conjunction with a different ARexx program might become an automated BBS program, a BBS itself, or perhaps, with more modules, a game or series of games that may be played via modem. How about a terminal program with a text area and a drawing area? A simple paint program could become a sophisticated graphic terminal program.

Last, but not least, the ARexx package includes a library of high level functions you can call from your own programs. There are functions for handling string operations, file manipulation, number conversions, message passing, resource tracking, dos packets, and more. Complete programmer information for using this library is included in the package, along with instructions and hints for using it, for writing an ARexx interface into your program, and even for writing your own function libraries for extending the ARexx language (one person has written a serial port function library already).

ARexx was written by William Hawes, who is also the author of Conman, a shareware CLI enhancement program. He has also written WShell, a very nice companion program to ARexx.

I rate ARexx as one of the best programs I have ever purchased for my Amiga, and in combination with WShell and Conman, I have an easily configurable environment that is hard to beat. I would even go so far as to say that ARexx may be the single most important program yet to arrive for the Amiga, and that it will have a significant effect on the way we use the machine.

As you might have guessed by now, I recommend it wholeheartedly.

*ARexx is available from your local dealer, or directly from:*

*William Hawes*
*P.O. Box 308*
*Maynard, Massachussets 01745*
*(617) 568-8695.*

SEX VIXENS
FROM SPACE
Adult graphic adventure game for the Amiga™ and IBM™ computers.
You have been assigned to a high priority mission by the Federated Government. In order to save the galaxy, you must locate and destroy a deadly weapon.
See your dealer or call 1-800-552-6777. In Illinois call 312-352-7323.
Free Spirit Software Inc.
905 W. Hillgrove, Suite 6, LaGrange, IL 60525
Suggested retail price $39.95.

# COMSPEC'S SEVEN BITS TO SPICE UP YOUR BYTES

**MCS 1000**
Connect any IEEE type printer to virtually any computer utilizing standard Centronics output. Plugs into a parallel port on your computer.

**AX 2000**
1 or 2 MB FAST RAM for the Amiga 1000 computer. Provides the extra RAM necessary to run advanced applications quickly and efficiently. Runs with no wait states. Complete with non-volatile RAM disk.

**SD-20 SD-40 SD-60**
Auto-booting external hard drives for the Amiga. The Comspec Hard Drive system consists of a SCSI Host Adaptor * and a Hard Drive Chassis, available in 20, 40, and 60 MB sizes. The only Hard Drive system for the Amiga that provides auto-booting under WB 1.2.

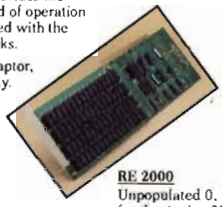
* Also available separately.

**BN 20**
Bernoulli 20MB 5 1/4" removable cartridges allow you to store sensitive data in a secure place. Also allows for the sharing of large amounts of data between computers. Provides the storage space and speed of operation of a hard drive combined with the portability of floppy disks.

Requires SCSI host adaptor, also available separately.

**MCS 8000**
Multi-user printer network/buffer which allows one printer to do the work of eight. Fully compatible, the MCS 8000 allows up to eight computers to share one printer. A must for any multi computer environment.

**MCS 6550**
256K print buffer. Frees up computer time by allowing you to process and print at the same time. Works alone or can be combined with the MCS 8000 to increase memury. No installation is required.



**RE 2000**
Unpopulated 0, 1 or 2 MB FAST RAM for the Amiga 2000. Similar to the AX2000 with 1/6 power consumption of competitive products. This highly reliable memory board is compatible with all standard Amiga hardware and software.

**Available from:**

**COMSPEC**

74 Wingold Ave., Toronto, Ont., M6B 1P5 (416) 785-3553 FAX 785-3668

**Distributed worldwide by:**

**Precision Software Ltd.**
6 Park Terrace
Worchester Park
Surrey KT4 7J2
England
01 330-7166
Fax 01 330-2089

**Microton**
Bahnofstrasse 2.
Postfach, 40 Ch 2542
Pieterlen, Switzerland
032 872429
Fax 032 872482

**Data Stream**
29 Caryle St.
Naiper, New Zealand

**Run Informatique**
62 Rue Gerrard
Paris, France
45 81 5144

**Airstat of Canada Inc.**
3575 — 14th Ave.
Markham, Ont. Canada
L3R 2L6
1-800-263-3314
Fax (416) 470-0805

**Interest A.S.**
Markveien 44B
P.O. Box 59
4030 Hinna, Norway
(04) 881039
Fax (04) 881969

*Specifications subject to change without notice.